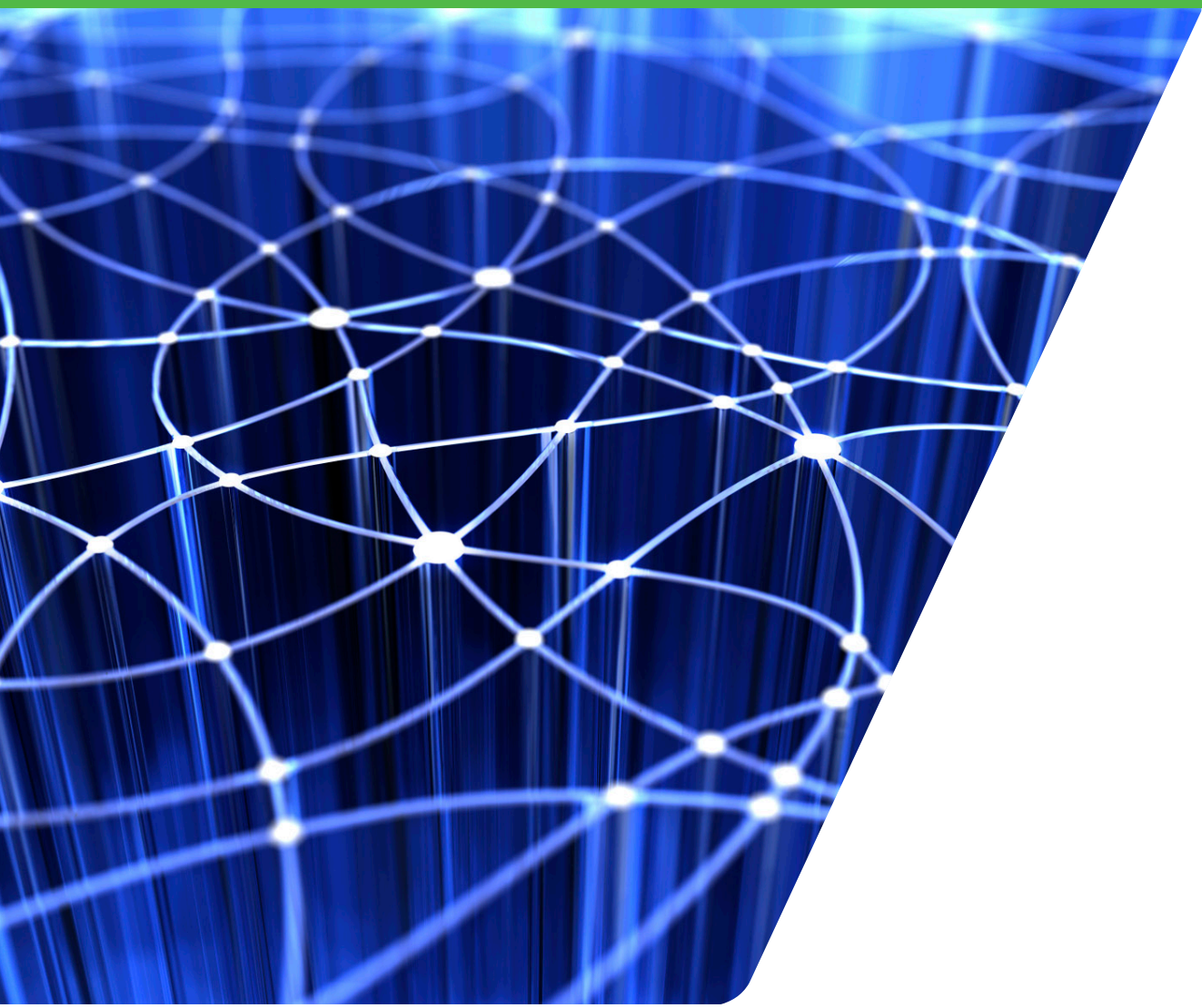
VTT PUBLICATIONS 751



Sakari Stenudd

Using machine learning in the adaptive control of a smart environment

VTT PUBLICATIONS 751

Using machine learning in the adaptive control of a smart environment

Sakari Stenudd

ISBN 978-951-38-7420-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

Sakari Stenudd. Using machine learning in the adaptive control of a smart environment [Koneoppimisen käyttö äly-ympäristön mukautuvassa ohjauksessa]. Espoo 2010. VTT Publications 751. 75 p.

Keywords smart space, inter-operability, control loop, adaptive systems, self-adaptive software, reinforcement learning, Smart-M3 IOP

Abstract

The purpose of this thesis is to study the possibilities and need for utilising machine learning in a smart environment. The most important goal of smart environments is to improve the experience of their inhabitants. This requires adaptation to the behaviour of the users and the other changing conditions in the environment. Hence, the achievement of functional adaptation requires finding a way to change the behaviour of the environment according to the changed user behaviour and other conditions. Machine learning is a research area that studies the techniques which make it possible for software agents to improve their operation over time.

The research method chosen in this thesis was to review existing smart environment projects and to analyse the usages of machine learning within them. Based upon these uses, a model for using machine learning in a smart environment was created. As a result, four different categories of machine learning in smart environments were identified: *prediction, recognition, detection* and *optimisation*. When deployed to the environment, these categories form a clear loop structure in which the outputs of previous learning agents serve as inputs for the next agents, which ultimately enables the making of changes to the environment according to its current state. This kind of loop is called a *control loop* in adaptive systems.

To evaluate the suitability of the model for using machine learning in a smart environment, two demonstrations were carried out in an environment using a Smart-M3 inter-operability platform, both utilising machine learning in one of the above-discussed categories. In the first experiment neural networks were used to predict query latencies in different situations in the environment. The predictions of the network were compared to the outputs of two simpler models. The results showed that the neural network approach was capable of adapting to rapid changes more quickly. However, it also made more false assumptions about the impact of the different parameters.

The second experiment belongs to the optimisation category. In this experiment a decision maker was implemented for a resource allocation problem in a distributed multi-media streaming application. It used reinforcement learning with a look-up table and an implementation of the Q-learning algorithm. After the learning period the agent was capable of making optimal decisions.

The experiments confirm that it is suitable to use the model described in this thesis in smart environments. The model includes the most important uses of machine learning and it is consistent with other results in the areas of smart environments and self-adaptive software.

Sakari Stenudd. Using machine learning in the adaptive control of a smart environment [Koneoppimisen käyttö äly-ympäristön mukautuvassa ohjauksessa]. Espoo 2010. VTT Publications 751. 75 p.

Avainsanat smart space, inter-operability, control loop, adaptive systems, self-adaptive software, reinforcement learning, Smart-M3 IOP

Tiivistelmä

Opinnäytetyöni tarkoitus on tutustua koneoppimisen käyttömahdollisuuksiin ja -tarpeisiin älykkäässä ympäristössä. Älykkäiden ympäristöjen tärkein päämäärä on niiden käyttäjien käyttökokemuksen parantaminen. Tämä vaatii mukautumista käyttäjien käytökseen sekä muihin muuttuviin tilanteisiin ympäristössä. Mukautumisen saavuttamiseksi tarvitaan tapa muuttaa ympäristön toimintaa tapahtuvien muutosten mukaan. Koneoppiminen on tutkimusalue, joka käsittelee sellaisia tekniikoita, joita käyttäen ohjelmistoagentit voivat parantaa toimintaansa ajan kuluessa.

Opinnäytetyön alussa tutustutaan olemassa oleviin äly-ympäristöprojekteihin ja tarkastellaan niissä käytettyjä koneoppimismenetelmiä. Käytettyihin menetelmiin perustuen esitetään malli, joka kuvaa, miten koneoppimismenetelmiä voidaan käyttää älykkäissä ympäristöissä. Malli sisältää neljä eri koneoppimistyyppiä: *havainnointi*, *tunnistaminen*, *ennustaminen* ja *optimointi*. Kun näitä tyyppejä käytetään äly-ympäristössä, ne muodostavat selkeän silmukkarakenteen, jossa seuraavat oppivat agentit voivat käyttää edellisten tuloksia. Tämä mahdollistaa lopulta sen, että ympäristöön voidaan tehdä muutoksia sen nykyisen tilan perusteella. Tällaista rakennetta kutsutaan mukautuvien järjestelmien alueella nimellä *ohjaussilmukka*.

Jotta voitaisiin arvioida luodun mallin soveltuvuutta, luotiin kaksi mallin osa-aluetta käyttävää demonstraatiota käyttäen Smart-M3-yhteentoimivuusalustaa. Ensimmäisessä toteutuksessa käytettiin neuroverkkoja ennustamaan kyselyjen viivettä erilaisissa äly-ympäristön tilanteissa. Neuroverkon ennusteita verrattiin kahden yksinkertaisemman mallin tuloksiin. Testit osoittivat, että neuroverkkomenetelmä pystyi mukautumaan nopeisiin muutoksiin aiemmin, mutta se teki myös joitakin vääriä olettamuksia eri parametrien vaikutuksesta tulokseen.

Toinen koe kuuluu optimointiluokkaan. Siinä toteutettiin päätöksentekijäohjelma, jonka tuli ratkaista resurssien kohdentamisongelma hajautetussa multimedian suoratoisto-ohjelmassa. Päätöksentekijässä sovellettiin vahvistusoppimis-

tekniikkaa käyttäen hakutaulukkoa ja Q-oppimisen toteutusta. Oppimisjakson jälkeen agentti pystyi tekemään optimaalisia päätöksiä suurimman osan ajasta.

Tehdyt kokeet osoittivat, että työssä kuvattu malli sopii käytettäväksi älykkäissä ympäristöissä. Malli kattaa tärkeimmät koneoppimisen käyttökohteet ja on yhtäpitävä muiden tulosten kanssa, jotka on saatu äly-ympäristöjen ja mukautuvien ohjelmistojen alueella.

Preface

This Master's thesis was written at the VTT Technical Research Centre of Finland in the Software Architectures and Platforms Knowledge Centre. The work was carried out as a part of the TIVIT/DIEM (Devices and Information Ecosystem) project.

I would like to express my sincere gratitude to my technical supervisor Senior Research Scientist Anu Purhonen for her support and valuable comments during the work. Further I would like to thank Research Professor Eila Ovaska and Senior Research Scientist Ville Könönen who have helped me with their expert feedback. I would also like to thank Professors Jukka Riekki and Janne Heikkilä who are the reviewers of this work at the University of Oulu.

Oulu, Finland 23 August 2010

Sakari Stenudd

Contents

Abbreviations

ACHE	Adaptive Control of Home Environments, a smart home system
ANN	Artificial Neural Network, a data representation model in machine learning
CRF	Conditional Random Field, a probabilistic model similar to HMM
ECA	Event-Condition-Action, a rule model for specifying actions in defined states
GA	Genetic Algorithm, a machine learning technique for searching optimal hypotheses by altering them
HMM	Hidden Markov Model, a machine learning technique that learns sequential data
IOP	See Smart-M3 IOP
IP	Internet Protocol, a communication protocol used in the Internet; it provides addressing capabilities to a network
KP	Knowledge Processor, an entity in Smart-M3 architecture that uses and/or produces information
MAPE-K	Monitor, Analyse, Plan, Execute, Knowledge, a control loop used in autonomic computing
MAS	Multi-Agent System, a way to reduce the complexity of a system by dividing it into smaller tasks and performing those tasks with individual agents
MDP	Markov Decision Process, a concept utilised in reinforcement learning. It requires that the state changes and rewards in the environment depend only on the current state and action.
MIT	Massachusetts Institute of Technology, a private research university located in Cambridge, Massachusetts, USA
ML	Machine Learning
NB	Naive Bayes, an assumption that the different features of the feature vector are conditionally independent
NoTA	Network on Terminal Architecture, provides a common communication protocol and module interfaces for embedded devices
Ogg	A free and open media container format
OWL	Web Ontology Language, a set of W3C recommended languages based on RDF and RDFS
RDF	Resource Description Framework, a way to represent information in the form of subject-predicate-object triples
RDFS	RDF Schema, a basic ontology language

RL	Reinforcement Learning, a machine learning technique in which the agent learns from possibly-delayed rewards instead of labelled examples
SE	Smart Environment, a physical environment that aims to improve the experience of its inhabitants by utilising knowledge about them and itself
SIB	Semantic Information Broker, an entity in Smart-M3 architecture that is used to store and deliver information
Smart-M3 IOP	Smart-M3 Inter-operability Platform, a smart environment platform that focuses on opening and sharing information from different domains, devices and vendors to be used by other entities
SVM	Support Vector Machine, a machine learning technique based on linear discrimination
TCP	Transmission Control Protocol, a networking protocol that provides reliable end-to-end connection over IP
TCP/IP	A set of communication protocols used in the Internet and other similar networks, named after the most important protocols in it (TCP and IP)
UDP	User Datagram Protocol, a networking protocol
UPnP	Universal Plug and Play, a service-level protocol set to connect and use different devices seamlessly
URI	Uniform Resource Identifier, a standard syntax for defining identifiers for abstract or physical resources
W3C	World Wide Web Consortium, a consortium that develops standards for the World Wide Web

1. Introduction

The increase in performance and decrease in size of computing devices, along with advances in other supporting fields, have augmented the amount of research conducted about smart environments in which devices embedded into the environment aim to improve the user experience [1]. There are already quite a few projects aiming to create such environments, for example ACHE [2], MavHome [3] and iDorm [4]. However, these projects have the main focus on creating a successful smart environment within one domain – such as a smart home. The Smart-M3 inter-operability platform (IOP) [5] is a more generic solution to communication between devices at the information level and thus enables the creation of smart environments.

Dynamics and complexity are very important characteristics in smart environments. Smart environments in different domains differ and even in the same domain the environment changes as new devices are introduced. In addition, the user behaviour and preferences may change [2]. Therefore it is difficult, if not impossible, to design algorithms that are able to control the environment in such a way that user comfort is maximised in every situation. This is why it is useful for the control of a smart environment to be *adaptive*. Studies on adaptive systems, autonomic computing and self-adaptive software state that adaptiveness helps to reduce the costs of handling the complexity of software systems and in handling unexpected and changed conditions [6].

By definition, a software agent is said to learn when its performance in a certain task improves with experience [7]. The machine learning research field studies the ability of software agents to learn. According to the definition, it may be suitable to use machine learning techniques in the adaptive control of smart environments. In fact, they are already used in self-adaptive software [6] and smart environments [8].

In this work, the Smart-M3 IOP as a new inter-operability solution was chosen to be used as a platform with which to create smart environments. The goal of this work was to evaluate the suitability of using machine learning techniques to achieve adaptive control in an environment using Smart-M3 IOP. Existing smart environment projects were studied in order to find the uses of machine learning and the ways to achieve adaptive control. Based on this, a model for using machine learning in Smart-M3 IOP is presented. The model was validated using two separate demonstrations.

This thesis starts by introducing smart environments generally and then specifically covering some interesting smart environment projects in Chapter 2. The next chapter (Chapter 3) gives a brief description of machine learning and studies the uses of machine learning in smart environments in more detail from a machine learning perspective, including some suitable techniques and algorithms for the identified problems used by researchers from other areas. In Chapter 4 the results from previous studies are combined and, based on them, a model for using machine learning in a smart environment using Smart-M3 IOP is presented. Chapter 5 describes the implementation of two cases with the aim of validating the model. In Chapter 6 the results of the cases are discussed and the contribution of this thesis evaluated. Finally, Chapter 7 concludes the thesis.

2. Smart Environments

This chapter defines the general features of smart environments and describes some existing smart environment implementations and projects. The discussion focuses on the machine learning uses within them.

2.1. Smart Environment

A smart environment (SE) can be defined as an environment that 'is able to acquire and apply knowledge about the environment and its inhabitants in order to improve their experience in that environment' [1]. Therefore the environment must have some kind of sensors to be able to perceive its current state and the actions of the inhabitants and actuators in order to change its state. This section presents the characteristics of smart environments based on the requirements set for them and the features realised in the prototype solutions for different application domains. These summaries help in understanding the need for machine learning in smart environments.

Cook and Das defined the five general features of smart environments [8]:

1. *Remote control of devices.* Every device must be controllable remotely or automatically and must not require a dedicated user interface.

2. *Device communication.* The devices must be able to communicate with each other in order to build a model of the environment. They must also be able to gain access to external information sources such as the Internet.

3. *Information acquisition from intelligent sensor networks.* There must be a way to share the information gathered by the different sensors in the environment. Using this information, the environment can constantly adjust its state to better meet the requirements.

4. *Enhanced services by intelligent devices.* Information from sensor networks and communication with the outside world allow device manufacturers or programmers to create more intelligent devices that can add value to their functionality by using this external information.

5. *Predictive and decision-making capabilities.* The previously-described features allow the creation of smart environments. However, controlling this kind of environment manually would require constant monitoring and adjusting of the devices. To get the adjustments fully automated the devices themselves must be able to learn the optimal adaptation policies.

As mentioned in point five, predictive and decision-making capabilities require the devices to learn adaptation policies. Additionally in point three, information acquisition from intelligent sensor networks may benefit from machine learning techniques such as data mining, as the rest of this thesis shows. Machine learning has also been used in the control of some devices (point one above).

Solutions for smart environments have already been created in numerous research projects and some are presented in Section 2.2. Hermann et al. [9] listed the key aspects of the realised prototypes of smart environments as follows:

- Highly integrated and seamlessly available data, services and resources in public and private environments.
- The exchange of information, the access rights of objects, ambient resources and devices.
- The exchange of personal information between a number of users and the environment.
- The location-based availability of nearby entities, location-based UIs for services, data and applications.
- System 'intelligence': adaptivity and, to some degree, autonomous system decisions, e.g. on the use of ambient systems or data exchange.

The last item in this listing, system intelligence, states that in existing projects systems are typically adaptive and autonomous. The next section shows that this adaptivity and the autonomous decisions are implemented many times using machine learning techniques.

2.2. Existing Smart Environment Projects

This section presents a few existing projects with the goal of creating smart environments. In addition, the uses of machine learning techniques in the projects are described. Projects that include uses of machine learning were chosen to be presented and other projects that may otherwise be significant but do not concentrate on such things were excluded. As can be seen, most existing projects focus on building domestic environments such as smart homes.

2.2.1. ACHE

Mozer [2, 10] describes ACHE (Adaptive Control of Home Environments), an adaptive house that controls the comfort systems of a home such as lightning, ventilation and air and water heating. The objectives of ACHE are the prediction of inhabitant actions and the decrease of energy consumption. It tries to decrease the need for manual control of the systems by anticipating the need to adjust them. Figure 1 shows the architecture of an ACHE system. State transformation calculates some statistic values from the state information. The occupancy model determines which zones (rooms) are currently occupied in the house and predictors try to forecast how the state is going to change in the near future. The set-point generator determines the target value of the needed adaptation, for example the target temperature of the room, and the device regulator makes the actual adjustments by controlling the physical devices. ACHE has been deployed into a real house environment and it was able to reduce the need to explicitly adjust the systems under its control.

The three components shown at the top of Figure 1 (device regulator, set-point generator and predictors) are adaptive and thus use machine learning. The predictors use

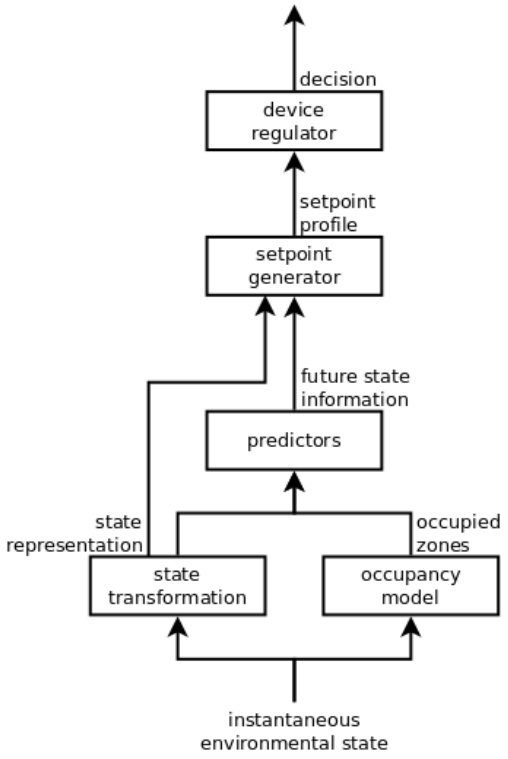


Figure 1. The system architecture of ACHE.

feed-forward neural networks and, in some cases, also look-up tables in combination to make predictions. Both the set-point generator and device regulator need to learn; the set-point generator tries to behave according to user preferences and the device regulator tries to find the optimal way to achieve the targets. Depending on the domain, the components can use, for example, reinforcement learning to directly locate good control actions or neural networks to create a model of the environment. [2]

2.2.2. MavHome

The MavHome (Managing an Intelligent Versatile Home) Project (e.g. [11, 3]) uses multi-agent systems (MAS) and machine learning techniques to create a home environment that is able to act as a rational agent. Figure 2 shows the architecture of MavHome. The architecture is divided to four abstract layers: Decision, Information, Communication and Physical. The Communication layer is used by the both of the higher level layers in the architecture. These abstract layers are realised by concrete functional layers which are also shown in Figure 2: Physical components, Computer interface, Logical interface, Middleware, Services and Applications. When a sensor in the environment makes a measurement, information flows from bottom to top. The Communication layer transmits the information to another agent if needed, components of the Information layer store the measurement into a database and may process it into a more useful form. The Decision layer receives the information if it is interested

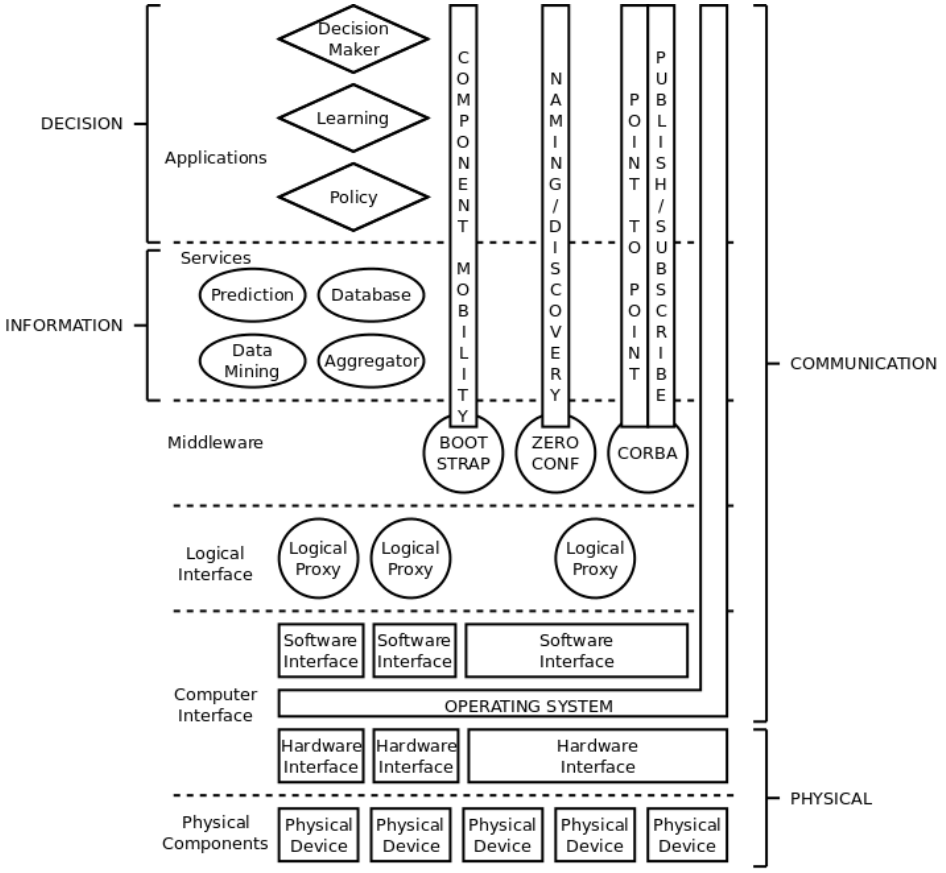


Figure 2. MavHome abstract and concrete architecture.

in it and can select a needed action which is updated to the database in the Information layer and delivered to the appropriate effector via the Communication layer.

The performance of MavHome has been evaluated using both simulation and real data in an apartment with a full-time occupant. The apartment contained 25 controllers and many sensors, for example for light, temperature and humidity. In the experiment only motion sensors and light controllers were used and the goal was to reduce the need for manual interactions with the lightning although it is also possible to use the system for other goals. Both the simulated and real application of MavHome showed a more than 70% reduction in interactions after a week of usage. [12, 3]

The operation of MavHome is divided in three separate phases: *Knowledge discovery and initial learning*, *Operation* and *Adaptation and continued learning*. In the first phase, there are several machine learning methods used. Data mining is used to find activity patterns from the observed data which are then used to build a hidden Markov-based model of the environment. A prediction algorithm is also trained using the observation data. In addition an episode membership algorithm, which calculates the probability of a set of observations belonging to a certain episode, is trained using the observation data and activity patterns. The second phase utilises the models and algorithms created in the first phase in order to make decisions about the needed actions. In the third phase, the model of the environment is constantly adjusted according to the feedback gained from the actions. Data mining is also used to find new patterns from

the observation data. If there is a significant change detected in the activity patterns the model is broken and the system goes back to the first phase, otherwise the system only runs the second and third phases. [11]

In addition to the previously-described machine learning uses, Jakkula, Crandall and Cook have added anomaly detection capabilities to MavHome. [13]

2.2.3. iDorm

The Essex intelligent Dormitory [14, 15, 4] is a test-bed for ambient intelligence and ubiquitous computing experiments. It is a room that contains furniture such as a bed, wardrobe, study desk, cabinet and computer. Thus the room is similar to a combined study and bedroom. However, the room and the furniture contain many embedded sensors for components such as temperature, occupancy, humidity and light-level sensors and actuators such as door actuators, heaters and blinds. The dormitory can also be monitored and controlled using a virtual reality system which shows the sensor values and allows the user to control the actuators. It also shows a visualisation of the room. The controlling is done using the Java interface of iDorm.

There are three different networks in the iDorm. Most of the sensors and actuators are connected to the Echelon LonWorks network while the rest are connected to the Dallas Semiconductor 1-Wire network. Both of these networks are connected to an IP (Internet Protocol) network using gateways. The computer in the room is also connected to the IP network. The controlling and monitoring of these components is done through the iDorm server which is a gateway between the sensors and actuators and the outside world. This gateway provides a UPnP (Universal Plug and Play) interface to the sensors and actuators [16]. There are three types of computational artefacts connected to the iDorm server from the outside: the most important is the *iDorm embedded agent* and in addition to that there is a mobile service robot and physically-portable devices such as a pocket PC and a mobile phone. [15]

The iDorm embedded agent contains the most intelligence in the dormitory. It receives the sensor values, computes appropriate actions using the learnt behaviour of the user as a reference and sends the actions through the network to the actuators. It learns rules from the behaviour of the user and also uses predefined rules to handle safety, emergency and economical issues. The rules learnt from user behaviour are dynamic and they can be added, removed or modified whenever the behaviour of the user, or the environment, changes. The different rule sets are handled by fuzzy logic controllers. The learning is based on negative reinforcement and occurs whenever the user expresses dissatisfaction by changing the actions that the embedded agent has carried out. [15, 16]

The recent work related to iDorm has included, for example, creating and coordinating multiple embedded agents in the dormitory that handle their own related sets of rules [17, 18]. There has also been work regarding the use of genetic algorithms in optimising the search for solutions [18, 19]. In addition to the simulation results, there have been a few real-data experiments including an inhabitant or even many inhabitants living in the dormitory [15, 17, 18].

2.2.4. ThinkHome

In a recent paper, Reinisch et al. [20] proposed a concept called ThinkHome to apply artificial intelligence to smart homes with the aim of reducing energy consumption. It includes a knowledge base (KB) that stores data about the environment in ontology form and a multi-agent system (MAS) that contains specific agents for different tasks. There are, for example, a user preference agent and a KB interface agent that delivers the information from the KB to the other agents. A group of mandatory agents is defined to make the ThinkHome environment work. ThinkHome is still in the conceptual phase and there are no actual implementations yet.

There are two agents in the systems that may contain learning capabilities in ThinkHome [20]. There is a control agent that uses certain strategy to decide on the optimal adaptations. These adaptations are made according to simple predefined rules or a machine learning technology can be used to obtain them. The other agent said to contain learning capabilities is the user agent which is responsible for delivering user preferences to the environment. It should be able to learn the habits and preferred environmental conditions of the user. Although not explicitly defined in the paper to contain learning capabilities, the context inference agent could also use machine learning to enhance its operation. As the name suggests, its purpose is to identify contextual information like situations and locations and the identities of the users.

2.2.5. Other projects

There are also many other projects containing some aspects of smart environments. The Oxygen project at the Massachusetts Institute of Technology (MIT) [21] and IBM's DreamSpace [22] focused on creating new, more natural ways of interacting with the environment. Philips's ExperienceLab is a research facility which has its main emphasis on following the behaviour and reactions of the test participants when interacting with the smart environment [23]. Microsoft's EasyLiving [24] project aimed to aggregate diverse input/output (I/O) devices so that they could be used seamlessly and dynamically.

The PlaceLab [25, 26] is a joint initiative of the House_n research group at MIT and a technology processing and commercialisation company called TIAX, LLC. It is a residential building equipped with a large number of sensors including microphones, cameras, sensors sensing the state of doors and drawers and positioning sensors. The goal of the PlaceLab is to allow researchers to systematically test and evaluate technologies in a natural setting using volunteer participants. There has been work on developing activity recognition algorithms which are trained and tested using datasets gathered from the PlaceLab [27]. In this particular experiment, decision trees were trained using annotated sensor readings for activities.

2.2.6. Summary

In Table 1 the projects described in this chapter and the machine learning uses in them are summarised. As can be seen, MavHome has the most identified uses of ML, which is because it was the most focused on using machine learning in the smart home.

The used solutions to the machine learning problems within these projects are presented at the end of the next chapter.

Table 1. ML uses in existing SE projects.

Project	**ML uses**
ACHE	State prediction Set-point generation Device regulation
MavHome	Activity pattern detection Activity prediction Episode membership recognition Environment model creation Anomaly detection
iDorm	User-behaviour learning
ThinkHome	Decision making User-behaviour learning
PlaceLab	Activity recognition

3. Machine Learning

This chapter will present some background about general machine learning paradigms. First, the need for prior information about the problem is summarised, then some general definitions and classifications of machine learning systems are presented. After that the Bayesian framework for machine learning is briefly introduced. The next section presents some different data presentations and supervised algorithms, then the concepts of reinforcement learning and unsupervised learning as well as some other relating areas are presented. Lastly the uses of machine learning techniques in smart environment projects and related problems are described.

This chapter aims to be an introduction to machine learning and to help in choosing and understanding suitable methods for given problems. The problems in this case are the potential uses of machine learning described at the end of this chapter and in the next chapter.

3.1. Prior Knowledge in Machine Learning

There are many different machine learning algorithms but none of them can be said to be better than any other according to two famous theorems. The *No Free Lunch Theorem* states that if there is no prior information about the problem, any two algorithms may perform equally well in solving the problem: there are problems where a constant or random output performs better than another more complex approach. If an algorithm performs well in one problem, there must be another problem in which it will perform badly. According to the *Ugly Duckling Theorem* it is impossible to say that any two different patterns would be more similar to each other than any other two (an ugly duckling and a beautiful swan are as similar to each other as are two beautiful swans as long as the swans differ somehow). As a result, there must be some prior knowledge (or good guesses or assumptions) of the problems in order to be able to measure similarity. [28]

These prior assumptions about the problem are sometimes called the *inductive bias*. Every algorithm has some kind of inductive bias implicitly added to the problem: it can be, for example, preference for the simplest possible representation that classifies the training data correctly. [7]

3.2. Definitions

A machine is said to *learn* if its performance at some defined task or tasks improves with experience. In other words, the machine or the system can change itself so that it does the same task or tasks better (or more efficiently) next time. [7, 29]

A *hypothesis* is an instance that belongs to a *hypothesis space*. A hypothesis space consists of all the possible representations for solving a problem, for example all possible weights of a neural network or all possible combinations of instances in concept learning. The problem for machine learning is to find the correct (or an approximately correct) hypothesis from the hypothesis space. [7, 28]

The task of inferring a boolean-valued function from examples belonging, or not belonging, to a concept is referred to as *concept learning*. Many algorithms for concept learning use a *general-to-specific* ordering of hypotheses. Hypothesis h_1 is more general than hypothesis h_2 if all instances that are classified as positive by h_2 are also classified as positive by h_1. So the most specific hypothesis classifies all instances as negative and the most general hypothesis classifies all instances as positive. [7]

There are at least two different classes of problems for which machine learning techniques are used. In *classification* problems the task is to classify an instance into one of a discrete set of possible categories. In *regression* problems the aim is to approximate a real-valued target function. Concept learning is a special classification problem in which there are two distinct classes. [7]

Overfitting is a central problem in machine learning. It means that the system learns the training examples 'too well' so that it performs less well when it is used with data that are not in the examples. Figure 3 is an example of a classification situation where overfitting may occur. The dots represent training examples from two different classes (squares and diamonds). They are represented with two attributes, the values of which are shown on the x- and y-axes. When a classifier learns the classifying function so that it takes all the training examples into account, it gets a complex function that classifies all the training examples correctly (the thin line). However, when the classifier is used in real situations, a simpler function (the thick line) may classify more instances correctly. [28, 7]

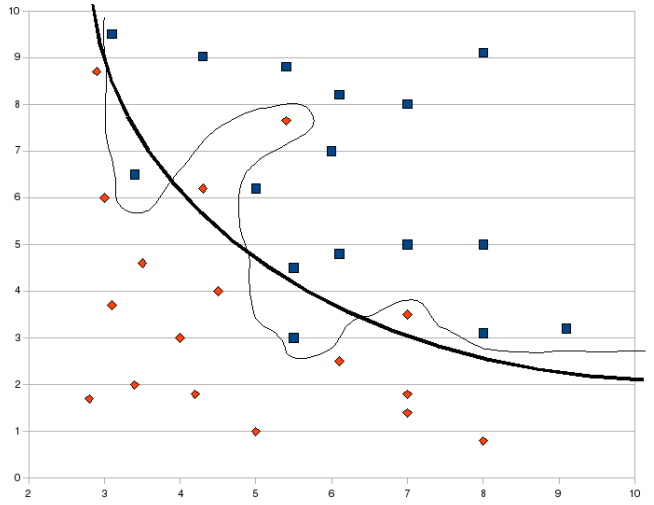


Figure 3. Training examples with an overfitted (thin line) and a desired (thick line) classifier.

3.3. Different Machine Learning Systems

Machine learning systems can be classified in many ways. In this section three classification dimensions are presented: *learning strategies*, *representations of knowledge* and *application domains*. [29]

The following learning strategies are arranged by the amount of inference needed by the learner: *rote learning*, *learning from instruction*, *learning by analogy*, *learning from example* and *learning from observation and discovery*. Rote learning means that there is no inference by the learner needed: it should only memorise the things as they are. Learning from instruction requires the integration of both new and prior knowledge. Learning by analogy requires some modification of the gathered knowledge to new situations. Learning from example is perhaps the most common used in research: the learner induces a general concept description from the given examples. Learning from observation and discovery (or *unsupervised learning*) needs most inference by the learner: no external teacher is present. [29]

There are many different possibilities in what type of knowledge the learner acquires. Some examples are: *parameters in algebraic expressions*, *decision trees*, *formal grammar*, *production rules*, *formal logic-based expressions*, *graphs and networks*, *frames and schemas*, *procedural encodings* and *taxonomies* [29]. The machine learning techniques presented in Section 3.5 are based on this kind of classification.

Learning algorithms have been applied to many different application domains, for example, to speech recognition, cognitive modelling, expert systems, natural language processing, music, sequence prediction and robotics [29]. The decision regarding which learning method to use is commonly made considering the application for which it is to be used but choosing an appropriate method is not always straightforward [30].

It is also possible to classify machine learning in three sub-sets using dependency on the teacher [28]. In *supervised learning* a teacher gives a set of labelled training examples from which the learner should generalise a representation. In *unsupervised learning* no information about the input is given and thus the system cannot know anything about the correctness of the outcome. The used algorithm forms clusters from the input data in a 'natural' way. *Reinforcement learning* lies between the two. No desired output is given, but the algorithm gets to know if the final output is correct or not. An example of this is an agent learning to play chess: it does not get feedback after every move but only receives the result of the game. Some supervised learning techniques are presented in Section 3.5, reinforcement learning is discussed further in Section 3.6 and unsupervised learning is introduced in Section 3.7.

3.4. Bayesian Reasoning

Bayesian reasoning is a probabilistic approach to pattern classification and machine learning. It provides a basis for many other learning algorithms and works as a framework for analysing the operation of other algorithms. In this approach the known (or guessed) prior (or *a priori*) probabilities and likelihoods of events are used to calculate the posterior (or *a posteriori*) probabilities. The basic formula used in Bayesian reasoning is called the *Bayes theorem* or *Bayes formula*:

$$P(\omega_j|x) = \frac{p(x|\omega_j)P(\omega_j)}{p(x)} \tag{1}$$

in which ω is called *the state of nature*, the class to which the pattern is to be classified. The symbol x denotes the feature vector that is used when classifying the samples. $P(\omega_j)$ is the a priori probability, $p(x|\omega_j)$ is the likelihood (that a sample of the class

ω_j has feature vector x) and $P(\omega_j|x)$ is the a posteriori probability. $p(x)$ is called *evidence* and it scales the shift from an a priori to a posteriori probability according to the probability of x. This can be written as

$$p(x) = \sum_{j=1}^{c} p(x|\omega_j)P(\omega_j) \tag{2}$$

when there are c different states of nature (categories). [28, 7]

If the parameters for the Bayes theorem are known it gives the exact probability of the class. However, often it is not feasible to compute the needed probabilities. Even in these cases Bayesian reasoning gives a way to analyse and understand the operation of other algorithms. There are also other algorithms that are directly based on Bayesian reasoning. [7]

3.5. Supervised Learning

In this section some of the best known supervised machine learning algorithms are discussed. The algorithms are organised by the way in which they represent the data they use in operation.

3.5.1. Naive Bayes model

In the Bayes theorem (Equation 1) the likelihood $p(x|\omega_j)$ is often difficult to determine and computationally unfeasible. Therefore it is often simplified with an assumption that the different features in the vector are conditionally independent of each other and depend only on the state of nature ω_j. So the likelihood can be written as:

$$p(x|\omega_j) = \prod_{i=1}^{n} p(x_i|\omega_j) \tag{3}$$

where x_i is the i:th feature in the feature vector x. This simplification is known as the *naive Bayes rule*. [7, 28]

The naive Bayes (NB) classifier is a very simple classifier. In the training phase it calculates the needed statistics $P(\omega_j)$ and $p(x_i|\omega_j)$ from the training data for every class ω_j and feature x_i. Then it classifies the data by calculating the probabilities of different states using the Bayes theorem and the naive Bayes rule. The NB classifier often works surprisingly well in practice. [7]

3.5.2. Decision trees

A *decision tree* is a representation of a learnt discrete-valued target function. Each node specifies a test of an attribute and each branch of a node corresponds to one value of the attribute. When an instance is classified, the attribute of the root node is tested and the corresponding path is followed. This is repeated for every subtree until a leaf

node (the classification) is reached. Decision trees are used, for example, to classify medical patients by their disease and equipment malfunctions by their cause. Generally decision trees are useful for problems with the following characteristics: [7]

- The instances are represented by attribute-value pairs.
- The target has discrete output values.
- The training examples may contain errors.
- The training data may contain missing attribute values (unknown values).

A basic algorithm for learning decision trees, ID3, constructs them from top to bottom by calculating the *information gain* of every attribute and thus always tries to find the attribute that best classifies the training examples [7]. An example of other algorithms for training decision trees is ID3's successor C4.5 [31]. A slightly newer approach is Random Forests [32]. This uses many decision trees initialised with random vectors and lets them vote for the result.

3.5.3. Linear discriminant functions

Linear discriminant functions determine a hyper-plane called a *decision boundary*. The decision boundary separates the different decision regions in the feature space. The linear discriminant function can be written as:

$$g(x) = \omega_0 + \sum_{i=1}^{d} \omega_i x_i \tag{4}$$

where ω_i are the components of the weight vector w and x_i are the input feature components. This function can be generalised by writing:

$$g(x) = \sum_{i=1}^{\hat{d}} a_i y_i(x) = a^t y \tag{5}$$

where a_i are the weight vectors and y_i are arbitrary functions of x, sometimes called ϕ functions. Function 5 is no longer linear in x but is linear in y. If $\hat{d} > d$ the function makes a mapping to a higher-dimensional space. This allows it to discriminate classes that are linearly inseparable in the initial feature space, although this comes at the cost of more complex computations. [28]

Perceptrons

Perceptrons are very simple linear discriminant functions that can be used as a basis to create the basic units of artificial neural networks. A perceptron calculates the linear combination of a vector of real-valued inputs and then outputs 1 if the result is greater than a certain threshold and -1 otherwise. The output $o(x_1, \ldots, x_n)$ is computed by

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases} \tag{6}$$

in which x_i is an input, and w_i is the weight of the input. The value of x_0 is always 1 and $-w_0$ is the threshold value of the perceptron. The summation can also be represented as the dot product of the input and weight vectors: $\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$. [7]

A single perceptron can be used to classify patterns that are *linearly separable* which means that it must be possible to separate them with a hyper-plane with an equation of $\vec{w} \cdot \vec{x} = 0$. A perceptron can be used to represent the primitive Boolean functions AND, OR, NAND and NOR, which can be used to create a network to represent any Boolean functions. For example, the XOR function (which is an example of linearly inseparable classes) can be implemented with the AND, NAND and OR functions, which requires three perceptrons. [7]

The simplest training algorithm for perceptrons is the *perceptron training rule*, which changes the weight associated with an input towards the desired output:

$$w_i \leftarrow w_i + \eta(t - o)x_i \tag{7}$$

where t is the target output, o is the output generated by the perceptron and η is a positive constant, the *learning rate*. The value of the learning rate is usually quite small or decreases as the number of iterations increase. The perceptron training rule works when the training examples are linearly separable. [7]

Support vector machines

Support vector machines (SVMs) are linear discriminant functions that map the feature space to a higher dimension. Training an SVM causes it to find the optimal hyperplane which has the maximum distance from the nearest training patterns, called *support vectors*. This is expected to give more generalisation capabilities to the SVM. [28]

Training an SVM requires choosing the ϕ-functions that map the input to a higher-dimensional space and using, for example, quadratic programming optimisation techniques to find the optimal hyper-plane. The ϕ-functions are often chosen using the designer's knowledge of the problem domain. Training an SVM is quite efficient and they can represent complex non-linear functions. [28, 33]

3.5.4. Artificial neural networks

Artificial neural networks (ANNs) are a practical way to approximate real-valued, discrete-valued and vector-valued target functions. ANNs are useful in many applications such as speech recognition, visual scene interpretation and robot controls. ANNs, using a training method called Backpropagation, are appropriate for problems with the following characteristics: [7]

- Instances are represented by many attribute-value pairs.
- The target function may be real-valued, discrete-valued or vector-valued.
- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the target function is required.

- The learnt target function does not have to be understandable by humans.

ANNs consist of sets of simple units that take a number of real-valued inputs and produce a single real-valued output. The units are inter-connected, so the output of one unit can be an input for another unit. The units with output that is not visible outside the network are called *hidden units*. The network can be cyclic or acyclic and directed or undirected but the majority of applications use directed acyclic ANNs. [7]

The units of ANNs can be, for example, perceptrons as described earlier. However, in many applications it is more practical to use other types of units. Examples of these are *linear units* or unthresholded perceptrons with output that is a linear combination of inputs and *sigmoid units* where the output o is:

$$o = \sigma(\vec{w} \cdot \vec{x}) \tag{8}$$

where $\vec{w}$ is the weight vector, $\vec{x}$ is the input vector and:

$$\sigma(y) = \frac{1}{1 + e^{-ky}} \tag{9}$$

in which the variable k determines the steepness of the function curve. Linear and sigmoid functions are differentiable and therefore it is possible to train them using a *gradient descent* to adjust the weights so that the errors are reduced most. [7, 34]

There are three different classes of neural network architectures. The *single-layer feed-forward network* is the simplest form of an ANN. It has an *input layer* of source nodes that does no computation but delivers the inputs to the *output layer* of neurons. Figure 4a is an example of an ANN of this type. In *multi-layer, feed-forward networks* there are also layers of hidden units. This kind of network can extract higher-order statistics as opposed to single-layered ones. An ANN is said to be *fully connected* if every node in each layer is connected to every node of the next layout. Otherwise it is said to be *partially connected*. Figure 4b shows an example of a fully-connected, two-layer ANN. The third class, illustrated in Figure 4c, is *recurrent networks* which have *feedback loops*. The feedback loops involve *unit-delay elements* (z^{-1} in the figure) which can result in non-linear dynamic behaviour. [34]

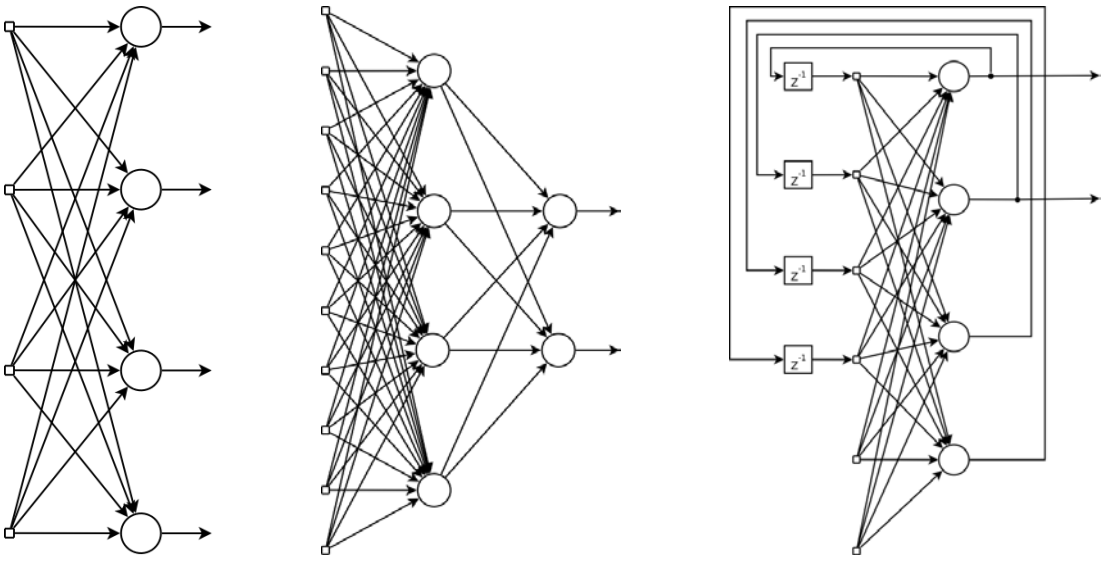
Perhaps the best-known training algorithm for neural networks is *backpropagation*. This can be used to train a network with a fixed set of units and connections. In the training phase the training examples are fed into the network and the error terms for the units are calculated. The calculations are made starting from the output units so that the error of the output is propagated to the previous layers in proportion to the weight of the connection. The weights are then updated to minimise this error. [7]

Although backpropagation is the most widely known algorithm in neural networks, there are also other possible learning algorithms. For example, Cascade-Correlation doesn't train a network with fixed topology but it does add new hidden units to the networks while training. The weights of the added units are not changed afterwards, but the output unit weights are changed repeatedly. This algorithm learns very quickly and there is no need to determine the number of hidden units before learning. [35]

3.5.5. Hidden Markov models

Hidden Markov models (HMMs) can be used when making sequences of decisions in cases where the decisions in time t depend on the parameters in time $t-1$ [28]. HMMs are widely used, for example, in speech recognition and gesture recognition applications [28]. An HMM has a finite number N of states $Q = \{q_i\}$. At each time t a new state is entered depending on the state transition probability $A = \{a_{ij}\}$ of the previous state. At each state the HMM produces an output symbol from the symbol set $V = \{v_k\}$ according to the observation probability distribution $B = \{b_{jk}\}$ of the state. An HMM is defined by these two probability distributions and initial state probability distribution $\pi = \{\pi_i\}$. The state of an HMM is not directly observable but it can be deduced using the observed symbols and probability distributions. [36]

There are three key issues or 'problems' in HMMs that must be solved in order to use HMMs in real applications. The first is the *evaluation problem*: Given the HMM, determine which is the probability that a particular observation sequence has been produced by the HMM. Secondly, there is the *decoding problem*: Given the HMM and the observations, determine the most likely sequence of states that created the observations. The last problem is the *learning problem*: Determine the parameters of HMM according to a set of training observations to maximise the probability that the observations are created by that model. There are solutions to each of these problems, for example, the *forward-backward procedure* is a solution to the first, the *Viterbi algorithm* to the second and the *Baum-Welch method* to the third. [36, 28]



(a) A single-layer feed-forward network.

(b) A multi-layer feed-forward network.

(c) A recurrent network.

Figure 4. Different neural network types.

Figure 5 shows a simple HMM with three states $Q = \{q_1, q_2, q_3\}$ and three observation symbols $V = \{v_1, v_2, v_3\}$. The corresponding probability distributions are $A = \{a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}\}$, $B = \{b_{11}, b_{12}, b_{13}, b_{21}, b_{22}, b_{23}, b_{31}, b_{32}, b_{33}\}$ and $\pi = \{\pi_1, \pi_2, \pi_3\}$.

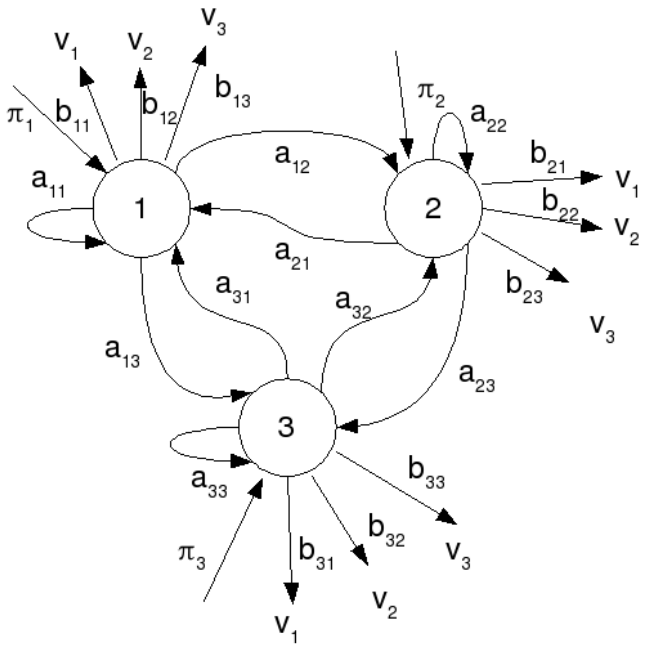


Figure 5. A simple HMM.

3.5.6. Instance-based learning

Instance-based learners do not create a new representation of the training data, they just store the data. The calculation is done when a new instance needs to be classified. Instance-based learners need much more storage space than other learners and may need much calculation in the classification phase. An advantage is that each new instance can be classified locally by only taking into account the training samples that are needed. Instance-based learners are sometimes called *lazy learners*. [7]

k-Nearest-Neighbour learning

k-Nearest-Neighbour learning is perhaps the simplest instance-based learner there is. All training and classification instances must correspond to points in an n-dimensional feature space $\mathbb{R}^n$ so that an instance is described by the feature vector

$$\langle a_1(x), a_2(x), \ldots, a_n(x) \rangle$$

where $a_r(x)$ is the value of the r:th feature of the instance x. The Euclidean distance $d(x_i, x_j)$ of two instances x_i and x_j is

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2} \,. \tag{10}$$

The k-nearest neighbour algorithm finds the k nearest instances to the classification instance from the training set and gives the most frequent value of them to the classification instance. The value k is usually a small odd number, for example three. [7]

3.5.7. Genetic algorithms

Genetic algorithms (GAs) are a set of learning algorithms that operate with populations of hypotheses which are used to generate new generations of population. Genetic algorithms are motivated by biological evolution and they use operations such as random mutation and crossover to change the hypotheses. The information for GAs is typically expressed as bit strings. For example decision trees can be encoded as bit strings: so genetic algorithms can be used to train decision trees. [7]

The use of GAs requires finding the best hypothesis from the current population. This is made with a *fitness function* that estimates the fitness of a hypothesis. Some of the most effective hypotheses are typically moved to the new population intact while the others are used to create a new offspring of hypotheses by using crossover and mutation operations on them. Genetic algorithms have been shown to be able to produce results comparable to other machine learning methods. [7]

3.5.8. Learning rules

Rules are very easy for people to read and understand. It is possible to, for example, train a decision tree and interpret it as a set of rules or search a satisfying rule set using genetic algorithms. However, there are also algorithms that directly learn the rule sets. They have two advantages compared to the previous methods: they can learn first-order rules which are more expressive than propositional rules and they can grow the rule set incrementally, one rule at a time. [7]

3.5.9. Summary of supervised machine learning methods

This section summarises the methods that are described above and some general characteristics of different representations and algorithms are given. The summary is based on this chapter and other literature [37, 38, 28].

Input data type: The input data (or feature values) can be discrete or continuous. Neural networks and SVMs usually perform well with continuous features and decision trees, rule learners and naive Bayes classifiers are good for discrete features. Instance-based learners are typically not directly suitable for discrete features.

Output data type: Neural networks can produce discrete-valued, real-valued and vector-valued outputs. Decision trees and naive Bayes classifiers produce only discrete-valued outputs.

Amount of training data needed: Neural networks and SVMs usually need a large amount of training data to learn while naive Bayes classifiers need only a relatively small data set.

Overfitting: Algorithms with few parameters to adjust tend to be less likely to overfit their behaviour. Neural networks, SVMs and decision trees are more vulnerable to overfitting than naive Bayes classifiers.

Multi-collinearity and non-linearity: Decision trees perform less well than artificial neural networks when the input data are highly correlated. ANNs can find a solution even when there is a non-linear relationship between the input and output features. HMMs can be used when temporality must be also considered (when the outputs also depend on the previous decisions).

Training time: The naive Bayes classifier trains very fast because it only needs a single pass on the training data. Decision trees are also quite fast to train but neural networks and SVMs are usually very slow. Instance-based learners need no training. Genetic algorithms usually need a large number of iterations in order to find suitable solutions.

Storage space: Most of the representations described in this thesis create a simplified model of the training data and usually do not require much storage space in the execution phase. Instance-based learners do not analyse the data until the result is needed, so they need to have all the training examples in the memory.

Missing feature values: In decision trees, neural networks and instance-based learners the missing values must be estimated or whole examples must be dropped from the training data whereas naive Bayes classifiers are able to simply ignore the missing values.

Irrelevant feature values: Neural networks and kNN are very sensitive to irrelevant features and their presence can make using these techniques impractical.

Noisy feature values: Rule learners and decision trees tolerate some noise on feature values because of their pruning techniques whereas kNN struggles with noisy values.

Number of parameters: If the model has less tunable parameters it is easier to use and understand but more parameters allow better control over the process. Neural networks and SVMs have many parameters while naive Bayes classifiers have much less. The instance-based learner kNN only has the k parameter.

Understandability: The operation and results of neural networks and SVMs are difficult to understand in comparison with decision trees, rule learners and naive Bayes classifiers. The operation of the kNN is very intuitive but the results are sometimes quite difficult to understand.

The characteristics are summarised in Table 2. The columns contain estimated values for the characteristics of neural networks (ANN), support vector machines (SVM), decision trees (DT), naive Bayes classifiers (NB), hidden Markov models (HMM), k-nearest neighbour learners (kNN) and rule-based learners (Rule). In input and output data types 'C' means continuous, 'D' means discrete and 'V' means vector. The more stars ('$*$') a method has the better it is considered in relation to the feature. For example in 'Amount of training data' one star means that the model typically needs a lot of training data to be useful. However, in the case of parameter numbers, bullets ('•') are used instead of stars. The number of bullets indicates directly the number of parameters: it is not always better that the model has many adjustable parameters.

Table 2. An overview of supervised machine learning methods.

	ANN	SVM	DT	NB	HMM	kNN	Rule
Input data type	C	C	D	D	C	C	D
Output data type	C / D / V	D	D	D	D	D	D
Amount of training data	$*$	$*$	$**$	$****$	$**$	$****$	$**$
Overfitting	$*$	$**$	$**$	$***$	$**$	$***$	$**$
Multi-collinearity	$***$	$***$	$**$	$*$	$**$	$*$	$**$
Non-linearity	$****$	$****$	$**$	$*$	$**$	$*$	$**$
Training time	$*$	$*$	$***$	$****$	$**$	$****$	$**$
Storage space	$***$	$***$	$***$	$****$	$***$	$*$	$***$
Missing features	$*$	$**$	$***$	$****$	$**$	$*$	$**$
Irrelevant features	$*$	$****$	$***$	$**$	$**$	$**$	$**$
Noise	$**$	$**$	$**$	$***$	$***$	$*$	$*$
Parameters	••••	••••	•••	••	•••	•	•••
Understandability	$*$	$*$	$***$	$****$	$**$	$***$	$***$

It should be noted that this summary tries to find the characteristic features of methods and offers quite a narrow view of the area. Different algorithms have different features and can give better or worse capabilities to a model in some features. For example, in the case of neural networks (see Section 3.5.4), the Cascade-Correlation algorithm requires much less training time than backpropagation.

3.6. Reinforcement Learning

In reinforcement learning (RL), a learning agent doesn't have a training set of correct actions but must determine them using a *reward* that it gets from the environment. The agent can observe the *state* of the environment and has a set of *actions* that alter the state. After every action the agent gets a reward which can be negative, positive or zero. The agent must learn a *policy* to achieve its goal, which can be for example to maximise cumulative rewards. [7]

In reinforcement learning the agent must choose the strategy to follow: it can *explore* the environment or *exploit* the already known states and rewards. There are many situations in which it is not possible to explore the environment thoroughly and then

choose the best paths, for example, when the number of actions that the agent can take is limited. This problem is called the *exploration–exploitation trade-off*. [39]

3.6.1. Markov Decision Process

In a Markov Decision Process (MDP) an agent has a set A of actions and can perceive a set S of states. At each point of time t the agent perceives the current state s_t and performs the action a_t. The environment produces a reward $r_t = r(s_t, a_t)$ and switches to the next state $s_{t+1} = \delta(s_t, a_t)$. Both functions r and δ may be probabilistic but they only depend on the state s_t and action a_t. This is called the *Markov property*. The functions are not necessarily known by the agent. The agent should learn a policy $\pi : S \to A$ which is used to select the next action in the current state. The cumulative value achieved by using policy π from an initial state s_t can be defined as:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{11}$$

in which the sequence of rewards r_{t+i} is generated by selecting the action given by the used policy in every subsequent state: $a_{t+i} = \pi(s_{t+i})$. The policy function π may also have a probabilistic outcome. The constant γ ($0 \leq \gamma < 1$), the *discount factor*, determines the weight of delayed rewards compared to immediate rewards. The value given by the equation is called the *discounted cumulative reward* but there are also other definitions of total reward such as *average reward* and *finite horizon reward*. [7]

3.6.2. Learning policies

The *optimal policy* π^* is the policy that maximises $V^{\pi}(s)$ for all states s. The value function of the optimal policy in state s is denoted as $V^*(s)$. The target is to learn the optimal policy. There are two different approaches to learning it: model-based and model-free. In model-based learning a model of the environment is learnt. That means learning the state transition function $\delta(s_t, a_t)$ and the reward function $r(s_t, a_t)$. When these functions are known, it is possible to solve the optimal action in every state. In model-free learning the model of the environment is learnt implicitly when the value of the different states is learnt. [40, 7]

An example of model-free learning is Q learning, in which the Q function:

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a)) \tag{12}$$

is learnt for every state-action pair and used to calculate the optimal policy:

$$\pi^*(s) = \operatorname*{argmax}_a Q(s, a). \tag{13}$$

The update rule of the Q learning algorithm can be presented as:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \tag{14}$$

where $\hat{Q}_n(s, a)$ is the learning agent's estimate of the Q value for state s and action a in time n, α_n is the learning rate in time n and s' is the new state caused by action a in state s. Q learning is a special case of *temporal-difference* (TD) learning. Another similar learning algorithm is SARSA which uses a special exploring rule to choose the actions to be taken. [40, 41, 7]

These algorithms are guaranteed to find the optimal value when every state-action pair is visited infinitely. However, there is usually a need to generalise the learnt functions. Therefore a look-up table cannot be used in many real applications but it can be substituted, for example, with a neural network that learns to estimate the Q values based on the state-action pair. Another way is to use separate networks that take the state as input and output the Q value for every different action. A third method is to use one network that takes the state as an input and outputs Q values for every action. [7]

3.7. Unsupervised Learning

Unsupervised learning, also called clustering, is a learning method in which there is no teacher and thus the training samples are unlabelled. Of course this makes the learning problem much more difficult but there are a couple of situations where using unsupervised learning is appropriate. Duda et al. list five reasons to use them: [28]

1. Collecting and labelling a sufficiently large set of sample patterns for supervised learning can be costly.

2. For some problems it is beneficial to use unsupervised learning to find candidate groups from data before labelling them.

3. Continuous unsupervised learning can improve the performance of a classifier when the characteristics of patterns change over time.

4. Unsupervised learning can be used to find features which can then be used in categorisation.

5. Applying unsupervised learning in new data can give some insight to its structure and aid in its analyses.

There are quite a large amount of algorithms commonly used in unsupervised learning, some of which are based on supervised-learning algorithms, however, their descriptions have been omitted from this thesis. A good source of information on this is, for example, Duda et al. [28].

3.8. Research Areas that are Based on Machine Learning

Machine learning techniques are also used in other research fields. They can serve as an alternative solution to problems that can also be solved with, for example, statistical analysis. In this section two such fields, namely data mining and anomaly detection, are briefly introduced.

3.8.1. Data mining

Data mining is a separate research area from machine learning. However, machine learning techniques have a significant role in data mining and therefore it is worth mentioning. In addition, data mining is used in some existing smart environment projects.

The research area of data mining concentrates on finding useful information from large sets of data. Data mining is a multi-disciplinary field that combines results, for example, from statistics, artificial intelligence, pattern recognition, machine learning, information theory and data visualisation. The mined information can be for example correlations, patterns, trends or groups. Data mining is already widely used in many industries. [42]

Although supervised learning techniques are also used in data mining [42], unsupervised learning is also common [43], however, the uses mentioned in this thesis utilise unsupervised learning.

3.8.2. Anomaly detection

Anomaly-detection systems are used to monitor some entities in order to detect anomalous behaviour. This is done by comparing the current activities to a previously-created model of normal behaviour. An alert is created when there is a sufficiently large deviation from the norm. Anomaly detection is used, for example, in intrusion detection systems. Patcha and Park list some benefits of using anomaly detection in that domain. [44]

- Anomaly detection systems are capable of detecting insider attacks.
- The attacker cannot be certain which activities set off the alarm.
- Anomaly detection systems are capable of detecting previously-unknown attacks.
- Normal activity profiles are tailored to each different deployment environment.

However, there are also some drawbacks when using anomaly detection in intrusion-detection systems: [44]

- The system must be trained before deployment in order to find 'normal' profiles.
- It is challenging to create normal training profiles and inappropriate profiles degrade the performance of the detector.
- Anomaly detection systems typically generate false alarms quite often.
- Specific alarms can be difficult to associate with the events that trigger them.
- Malicious users can gradually train the system to accept anomalous behaviour as normal.

The techniques used in anomaly detection are known from the fields of statistics, machine learning and data mining. Both supervised and unsupervised machine learning methods can be used to train anomaly detectors. Examples of the techniques used are Bayesian networks, hidden Markov models, decision trees, genetic algorithms, neural networks and clustering techniques. [44]

3.9. Machine Learning in Existing Smart Environment Projects

This section describes the use of machine learning in the existing smart environment projects mentioned in Section 2.2 in more detail, concentrating on the machine learning techniques chosen to solve the problems in them. In addition, some solutions to similar uses from different sources are described in order to find different approaches to the same problems. The uses of ML with already-used methods are summarised in Table 3. The classification of the uses is based on the identified uses in Table 1.

Table 3. ML problems in existing SE projects and example methods for solving them.

Use	ML Methods (Project or Author)
Event prediction	Neural networks [2] Statistical model learning [11] Rule learning [45, 46] Genetic algorithms [46] Hidden Markov models [47]
Latency prediction	Artificial neural networks [48]
Activity pattern identification	Data mining [11, 45, 47, 49]
Activity recognition	Decision trees [26] Hidden Markov model [50] Conditional random fields [50] Naive Bayes classifier [51]
Anomaly detection	Data mining [13]
Device control	Neural networks [2] Reinforcement learning [2, 52]
Decision making	Neural networks [2] Reinforcement learning [11, 2, 53] Hidden Markov model [11] Rule learning, genetic algorithms [15]

3.9.1. Event and latency prediction

Event prediction was used in both the ACHE and MavHome systems. In ACHE neural networks were used to predict the subsequent state of the environment [2]. In

MavHome a statistical model that calculates the probabilities of different events (or episodes) was created [11].

There are also other approaches to event prediction. Vilalta and Ma [45] used an algorithm that learnt rules which were then used in predicting events. Weiss and Hirsch [46] used also rules for event prediction. Their rule set was learnt by using genetic algorithms. Laxman, Tankasali and White [47] used hidden Markov models trained using different episodes and related target events. These models were then used to find the likelihoods of the target events after the episodes.

Although not within smart environment domain, Ipek et al. [48] have created a well-performing solution to predict single programme execution time based on the inputs on one machine with only a negligible amount of noise caused by other processes. They used an artificial neural network for the problem. However, even such a simplified scenario required thousands of programme training runs to create a good model.

3.9.2. Activity pattern identification

In many cases, the training data for event predictors is created using a data-mining algorithm. In MavHome this algorithm is used to find activity patterns from sensor readings [11]. Vilalta and Ma [45] also used data mining to find sequences of events but the target events to be predicted were predefined. Laxman et al. [47] found the training values for the hidden Markov models using data mining. This kind of frequent-pattern mining has also been used in areas other than smart environments, for example, Han et al. [49] did a survey about technologies and uses for frequent-pattern mining.

3.9.3. Activity recognition

In an experiment within the PlaceLab [26] decision trees trained with the C4.5 algorithm were used to recognise activities. In addition to that, Van Kasteren et al. [50] have tested the suitability of hidden Markov models and conditional random fields (CRFs) in recognising activities. The CRFs used in the experiment are a probabilistic model that quite closely resembles HMMs, with the difference that the state transition probabilities are not represented as conditional probabilities but as *potentials* between two states. The experiments were done using a self-annotated data set with seven different activities to recognise and the apartment contained 14 digital state-change sensors. The results of the experiments showed a time-slice accuracy (the ratio of correct classifications to all classifications made) of about 95 % for both methods and a class accuracy (the average accuracy for all different classes) of 70–80 %.

Mühlenbrock et al. [51] used a naive Bayes classifier to detect activities. They used discretised sensor readings and other information such as the time of the day to detect one of the predefined activities. Their activity detector produced good results in simple cases where the activity induction from the inputs was quite straightforward.

3.9.4. Anomaly detection

MavHome also used anomaly detection. Jakkula, Crandall and Cook used temporal data mining on the observed activities in order to calculate the probabilities for the relations of events. When the probability of an event occurring is very small within a given time it is considered an anomaly. Similarly, if an event does not occur although its probability is high, it is considered an anomaly. The goal of their work is to support elderly people living at home for longer. Anomaly detection can help in this situation, for example, by notifying the system if the inhabitant has not taken the required medicine or has forgotten to switch off the stove. [13]

3.9.5. Device control

The ACHE system used neural networks and reinforcement learning to control devices [2]. This allowed the device controllers to learn how to achieve the target conditions, for example a target temperature.

Hafner and Riedmiller [52] used reinforcement learning to train a robot to allow fast and accurate control at arbitrary speeds. The robot had three omni-directional wheels arranged in a triangular shape and the problem was how to control them. They used a basic Q-reinforcement learning algorithm fitted to neural networks called Neural Fitted Q (NFQ). The system was able to control the wheels fast and accurately even when changing loads after less than five minutes of interaction with the robot.

3.9.6. Decision making

In the ACHE system the set-point generators were responsible for making decisions about controlling the devices. They used neural networks to create a model which was used to determine the target value of the controller or reinforcement learning to find the correct values directly. [2]

In MavHome the decision-making component used reinforcement learning with a temporal-difference learning algorithm. The model of the environment learnt by the decision maker was based on hidden Markov models. [11]

The iDorm learnt rules to find optimal actions in perceived states. The learning was based on reinforcement learning and the trigger for changing the rules was negative feedback from the user. For finding optimal rule sets genetic algorithms can be used. [15]

Prothmann et al. [53] have created an organic traffic control architecture which uses a rule-based, reinforcement-learning system to find the most effective way to control traffic lights in different situations. Their simulation results showed that this kind of architecture and learning system can substantially improve vehicle throughputs at busy intersections.

4. Model for Using Learning in a Smart Environment

This chapter describes a model for using machine learning in a smart environment. First, the Smart-M3 inter-operability platform used in this thesis as a platform for creating applications to smart environments is described. After that the machine learning uses described in the previous section are further elaborated and the model is created based on them. The use of the model in an environment using the Smart-M3 IOP is also discussed.

4.1. Smart Environment Inter-operability Platform

Devices and applications can inter-operate at a *device level*, *service level* and *information level*. Device-level interoperability gives devices the means to communicate and network with each other, for example antennas and TCP/IP protocol suite provide this kind of inter-operability. Service-level inter-operability technologies such as Universal Plug and Play (UPnP) and Network on Terminal Architecture (NoTA) can be used to discover and use different services provided by other devices. Smart-M3 IOP (also referred to as M^3) promotes information-level inter-operability in which the aim is to provide information without a need to know about interfacing methods to other entities. It can be used, for example, on top of NoTA or TCP/IP. [54]

Smart-M3 IOP defines a scalable producer–consumer infrastructure and a common information representation format. Different applications (which can be located in different devices) in the same application domain use a common predefined domain-specific *ontology* which defines the structure of the information they provide and use. If information is all the applications need, Smart-M3 IOP is a lightweight way to achieve inter-operability. For example, a simple application that shows the current outdoor temperature needs only the temperature information from the temperature sensor and perhaps the location information of the sensor. However, many devices also need inter-operability at lower levels. An example of this could be a counter application that counts visitors in a shopping area using video stream from a doorway [55]. In that case the device with the camera provides a service and the visitor counter uses the service. Smart-M3 IOP can be used to discover the service but the applications must still inter-operate at the service level and use a common video-streaming protocol.

4.1.1. Inter-operability in the Smart-M3 IOP

Smart-M3 IOP follows the blackboard architectural style combined with the publish–subscribe paradigm. The information-level view of Smart-M3 IOP is shown in Figure 6. The Semantic Information Broker (SIB) is the backbone of Smart-M3 IOP: it contains the information-sharing database and offers an interface to access and modify the information within. Knowledge Processors (KPs) can produce, modify or remove information in the SIBs. They can also subscribe to certain information in order to get a notification when it changes. The communication between KPs and SIBs is made

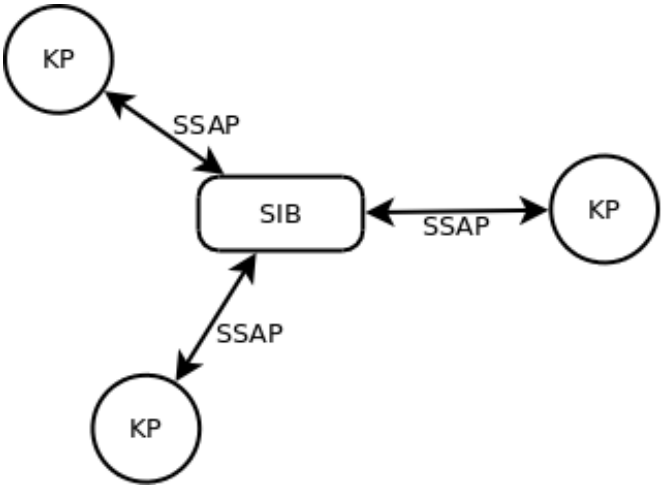


Figure 6. An information-level view of Smart-M3 IOP.

using a Smart Space Access Protocol (SSAP) which defines possible operations in the smart space. Table 4 shows these operations as described by Soininen et al. [5]. Different KPs and SIBs can be run in different processes or devices. A KP can be simultaneously connected to many SIBs and SIBs can be distributed.

A common understanding between knowledge processors is achieved using predefined ontologies. An ontology is a 'specification of a conceptualisation' and it defines a shared vocabulary, relationships between concepts and meanings and inference rules for them [56]. The standard language to represent ontologies is OWL (Web Ontology Language) [57] which is based on the RDF (Resource Description Framework) [58] data representation format and RDFS (RDF Schema) [59] language. Using the inference rules and semantics defined in an ontology it is possible to make reasoning

Table 4. SSAP operations.

Name	**Description**
Join	Begins a session between KP and SIB
Leave	Terminates the session
Insert	Inserts information into the smart space
Remove	Removes information from the smart space
Update	A combination of the remove and insert operations
Query	Queries information within the smart space
Subscribe	Sets up a persistent query
Unsubscribe	Terminates a persistent query
Results indication	Updates the result set of a persistent query
Unsubscribe indication	Notifies a knowledge processor of a smart space initiated termination of its subscription
Leave indication	Notifies a knowledge processor of a smart space initiated termination of the session

over the information and define mappings between different ontologies. In this case, reasoning means inferring information that is not explicitly defined in the database.

The RDF data representation format requires that all statements can be represented as triples which consist of a subject, a predicate and an object. An object can further be used as a subject in another triple. Therefore a collection of RDF statements can usually be presented as a set of directed, labelled graphs in which nodes are subjects and objects while predicates are represented as arcs. A node can be a URI (Uniform Resource Identifier, a standard syntax for defining identifiers) reference, a literal or a blank node. All predicates are URI references. A URI identifies a physical or abstract concept and one URI should be used to refer to only one thing. A blank node is an unnamed node that can be used as a subject or an object similarly to a URI reference, the only difference is that blank nodes are unnamed. A literal is used to identify values, for example numbers or ages. They are only used as objects in RDF. More information about RDF can be found in the W3C recommendation [58].

The reference implementation of Smart-M3 IOP [60] uses Wilbur library [61] in the SIB. It supports reasoning for an extended version of the RDFS language called RDFS++ when Wilbur Query Language queries are used [61]. If normal template queries defined in SSAP are used there is no reasoning support on the SIB side and thus it can be thought as only an RDF triple store.

4.2. Potential Uses of Machine Learning in a Smart Environment

In Section 3.9 the uses of machine learning in existing projects were summarised. This section aims to discuss their applicability to be used in a smart environment, especially in one using the Smart-M3 IOP.

The seven machine learning uses identified in the Section 3.9 can be further divided into four categories. Event prediction and latency prediction are *prediction* problems in which the goal is to create a model that can be used to decide on the most probable subsequent event. Activity recognition has some similar characteristics to the prediction from the machine learning perspective but here it is categorised as a *recognition* problem. It has the same goal of finding the most probable output, but it tries to recognise the current situation, not to predict coming ones. Activity-pattern identification and anomaly detection are *detection* problems in which the goal is to detect patterns occurring in the input data. These problems are typically solved using unsupervised learning techniques. The last category is *optimisation* problems which contains device control and decision-making problems. In these problems the goal is to find a policy that is optimal in the current situation.

The following subsections discuss these problems and how they fit into an environment using Smart-M3 IOP. Since inter-operability in Smart-M3 IOP is achieved using ontologies, the requirements for the used ontologies are also summarised here.

4.2.1. Detection

Detection uses for machine learning are, in some cases, supplementary solutions to the same problems that are solved using recognition algorithms. Detection problems are typically solved using data-mining algorithms. While recognition algorithms require the explicit labelling of training examples, data-mining algorithms divide the training instances into classes according to algorithm-specific criteria. Use of these algorithms can reduce the amount of work that would be done in labelling training examples for recognition algorithms. However, since the machine does not really know the semantics of the detected situations it may be more challenging to make conclusions based on these situations. For example predefined, rule-based reasoners cannot be used without mapping the detected situations to actual labels.

Anomaly detection is a special case of situation detection when the events are divided into two classes: normal and anomalous. This method could also be substituted with a recognition (concept learning) algorithm but the main advantage of anomaly detection, namely the detection of novel anomalies, would be lost.

As the name suggests, data-mining algorithms need a large amount of data to provide good results. To use data mining in Smart-M3 IOP environment ontologies must provide sufficient additional information about the data. For example the time of the observation is usually necessary to find observations that occur in the same time frame.

4.2.2. Recognition

Recognition problems are classification problems that are handled with supervised learning techniques. The most straightforward way to utilise recognition algorithms in a smart environment is to train the agent before deploying it to the environment. Online training in the environment is difficult because agent training requires labelled training examples and they are usually not available at runtime. However, in some cases it may be possible to get or deduce them if the output is right or wrong and thus improve the operation accordingly using, for example, reinforcement learning techniques.

The ontologies for the information that the recognition agents use must be defined but there are no special requirements for the design of the ontology. There must, of course, be a way to connect pieces of information that relate to the same activity so that the recognition can be based on this information.

4.2.3. Prediction

Common to all prediction problems is that their goal is to predict what is going to happen in the near future. The inputs for the predictors are not necessarily direct sensor readings but they can be pre-processed. Prediction problems can be classification or regression problems. An example of classification problems is event prediction, where the goal is to predict the most probable event or subsequent activity, while latency prediction is a regression problem in which the output – the latency value – takes on continuous values.

When prediction agents are used in a smart environment, online training may be a good approach. For example a latency predictor can obviously measure the actual latency it was predicting, compare it to the predicted value and improve its future predictions using the actual value as a training example.

When the predictors have already been trained it can be straightforward to apply them to Smart-M3 IOP. However, training the predictors using Smart-M3 IOP can be a more challenging problem. To achieve adaptive behaviour the possibility of using online training in some applications must exist. This requires that the information used in prediction can be coupled with the correct prediction output. If the examples are not removed from the database when new examples occur the pieces of information, for example sensor readings or statistics calculated from them, must be marked with the reading or calculation time. Training can be done incrementally or as batch training using a number of instances. In both cases the labels can be gathered by observing the environment.

4.2.4. Optimisation

Problems in which there is the possibility of gaining a reward from the environment and where the goal is to maximise the reward exist within the optimisation category. It is possible to use reinforcement learning for these problems. Some optimisation problems can be handled as prediction problems so that the rewards for different actions are predicted and the action with the highest reward would be chosen. However, the goal of optimisation problems is to find a policy that maximises long-term performance, not just selecting the best next action. Therefore an amount of experimenting with different actions may be advantageous.

Decision making is a good example of an optimisation problem. It requires taking into account many different variables and solving trade-offs between the benefits of different areas of the environment. Another example, device control, may not be present in every environment. It is needed when the devices are given high-level commands and they need to be changed to low-level actuator controls. This may require some experimentation in order to find the optimal way to control the actuators.

4.3. Interaction of Machine Learning Uses

Figure 7 shows the interaction of the four uses of machine learning. The figure is created based on the classifications in the previous section and the uses of ML in existing projects as described in Section 3.9. The arrow starting from the environment depicts sensor readings and the arrow ending at the environment depicts the control of actuators. In the figure detection and recognition categories are presented in one box: these are two different approaches to *pre-processing* sensor data. The pre-processed information can be used to predict forthcoming events. These predictions, along with the pre-processed sensor information, is then used to make decisions about how the state of the environment should be adapted. These decisions are then used to obtain the actual commands given to the actuators. The last two phases, decision making and device control, belong to the optimisation category. As can be seen, machine learning

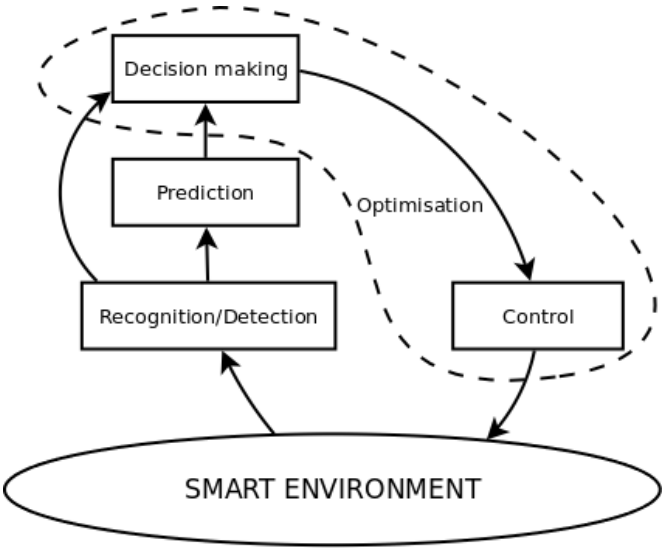


Figure 7. A model for using machine learning in a smart environment.

uses form quite a clear loop structure which enables the continuous adaptation of the application. This loop is called the *control loop*.

It is worth noting that all these phases can be realised without machine learning. Pre-processing can calculate some statistical value from the sensor readings. The prediction phase is not mandatory or it can use predefined rules. Decision making can also be done using fixed rules and control commands can be given directly or mapped using static mappings so that there is no need to learn them. However, in these instances there is the potential to achieve some improvement in the behaviour of the environment using machine learning.

Because Smart-M3 IOP allows the distribution of resources in the environment it is possible to have different components of the control loop in different devices as long as they have access to the same SIB. Figure 8 shows an example composition of a smart application using the control loop. In the figure recognition and detection tasks are shown as a pre-processor KP and the optimisation category has again been divided into the decision-making and device-control phases. The monitoring KP simply inserts sensor data into the SIB.

The figure actually contains three different control loops which define a hierarchical structure. Devices 1 and 2 both have a low-level control loop in which all the components are inside one device. Devices 3, 4 and 5 contain the components of a higher-level control loop that also includes KPs from Devices 1 and 2. Note that this loop does not directly include any monitoring or controlling KPs, although this would also be possible. In this application the higher-level control is thought to be controlling some parameters of the applications in devices, thus monitoring and controlling is thought to be done within the KPs of Devices 1 and 2.

5. Implementation

This section describes the implementation of the two demonstrated uses of machine learning in Smart-M3 IOP. The categories chosen for the demonstrations are prediction and optimisation. Section 5.1 describes an attempt to use ML for latency prediction. Section 5.2 describes the implementation of a simple decision maker that aims to learn how to make decisions that produce as the best results possible.

5.1. Latency Prediction

It is obvious that a poorly performing SIB can be a bottleneck and will limit the performance of a smart space using Smart-M3 IOP. In the future there will certainly be a wide variety of SIB implementations that can behave in different ways in different situations. However, the KPs cannot necessarily know a priori with which kind of SIB they are interacting. This case was motivated by the question as to whether it is possible to make some adaptations in the KP side in order to overcome the deficiencies in SIB implementation.

Using the Python version of SIB (not publicly available) it is apparent that the SIB performs worse when it is loaded. Notably, the number of subscriptions dramatically slows the operation of the SIB implementation. Although Smart-M3 IOP gives no guarantees about the time in which queries are processed, it may be beneficial for some applications to be able to adapt their behaviour according to the time taken by the processing of the operations. For example, if a sensor KP sends sensor information to the SIB twice per second and processing one insert operation takes 0.75 seconds, the SIB is not able to process all the requested operations. So the update interval should be lengthened.

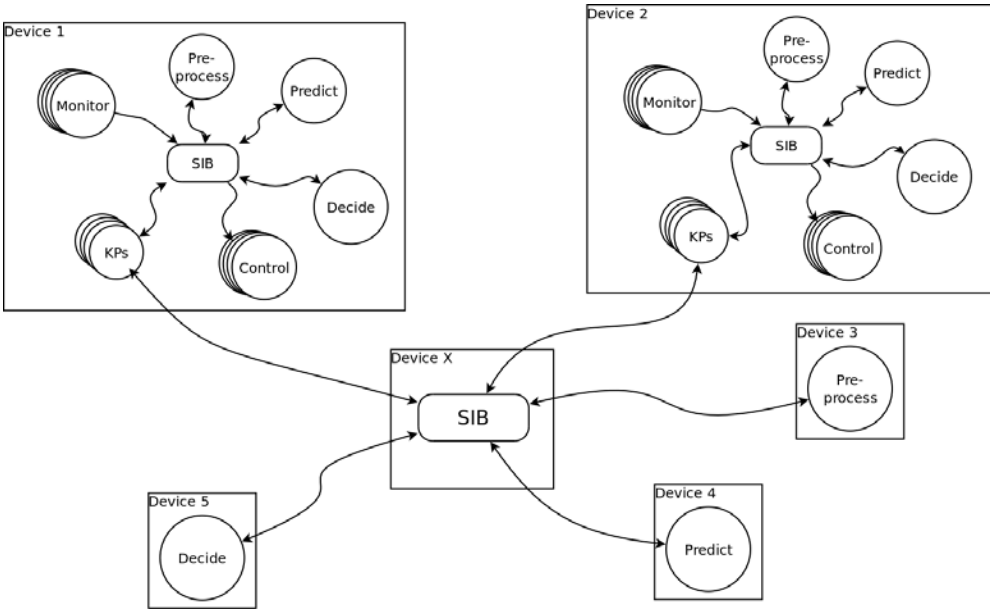


Figure 8. A smart application using a control loop.

The goal of this case was to discover if it is possible to model the behaviour of query latency using Python SIB and a machine learning algorithm with variables describing the load of the SIB as inputs. There are three parameters in an environment using Smart-M3 IOP from which it may be possible to infer the load level of the SIB. These are:

1. The number of joined KPs.
2. The number of triples inserted.
3. The number of subscriptions made.

It can be supposed that when there are more KPs joined to a SIB there is more activity and the SIB must perform more operations, which causes more latency. The amount of triples in the database may affect the latency because more data requires more searching. However, the amount of the increase in latency depends on the database implementation. The number of subscriptions seems to have the most affect on the latency. That is because the SIB must check if subscriptions require notifications after every insert, remove or update operation. In the implementation used the SIB it checks subscriptions even after query operations although this would not be needed. In addition to that, every subscription creates a separate thread for that particular subscription. This causes an excessive number of context switches, which again increases the latency.

5.1.1. Implementation

Currently, Smart-M3 IOP does not provide KPs with any information about the status of an SIB, for example the amount of triples, subscriptions or KPs. One possible way to get this information would be to make the required changes to the SIB code. However, changes to the SIB were avoided in the implementation to ensure the possibility of using the KPs with other SIBs as well. Thus, the only way to get this information is to explicitly insert it into the SIB. This requires that all the KPs that join to the SIB update the information in the SIB. However, because current SIB implementations do not provide atomic updates, this kind of cooperation is not possible. In the update operation KPs must define both the removed and inserted information. If two KPs try to update the same information at the same time, the updates are processed subsequently and as a result there would be two entries for the same data, which is not the desired outcome. So in this case this situation must be handled on the client side. In this implementation all the KPs are in a single process and the update operations are made in the same thread to avoid this kind of confusion.

Two separate KPs were implemented for the case. The first KP, RandomKP, generates load to the SIB. In fact, RandomKP generates many KPs that connect to the SIB but they are all implemented in one process. The other KP, PredictorKP, tries to predict the amount of time it takes for the SIB to process a query and compares it to the actual time. Both KPs were implemented using C++ language and the Qt toolbox[1] and their design is described in the next paragraphs.

[1] http://qt.nokia.com/

RandomKP

The RandomKP application has three main classes: RandomWidget, KPStarter and RandomKP. In addition, it uses a KPInterface class which wraps the C-language KPI_low interface [62] which is used to make operations to the SIB. RandomWidget is the user interface (UI) which allows the user to change and monitor operation rates. KPStarter is used to manage, create and delete RandomKP instances. RandomKP is the actual KP that interacts with the SIB. The class diagram of RandomKP is shown in Figure 9.

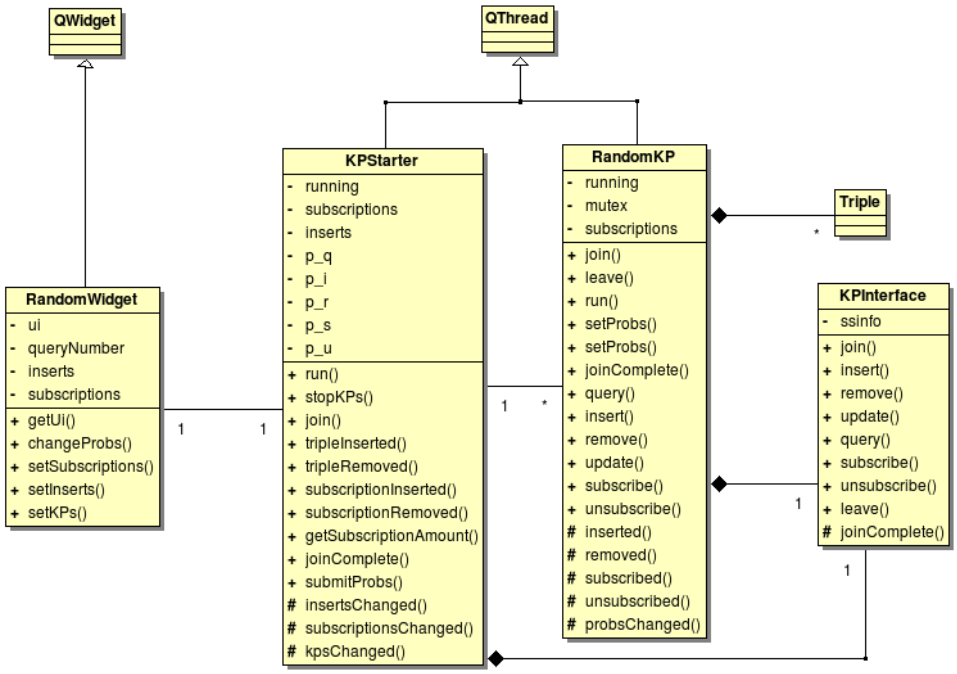


Figure 9. The RandomKP class diagram.

The UI of the application is shown in Figure 10. The ‘Amounts’ section shows the amounts of parameters that the application has created into the SIB. The ‘Start KP’ button allows the user to start simulation and ‘Stop KP’ stops it. The ‘Probabilities’ section shows the sliders that can be used to change the probabilities of different operations of the KPs and the probability of new KP creation. The values of these sliders are parameters of the other classes, KPStarter and RandomKP.

Figure 11 shows an activity diagram of RandomKP. KPStarter updates the three parameter values to the SIB and manages RandomKP instances. It runs in its own thread where the main loop starts new RandomKPs or deletes existing ones according to the value of the ‘KPs’ slider in RandomWidget which takes values from -100 to 100. A negative value means that the probability is that a KP will be deleted and a positive value means that the probability is that a new KP will be created. After the operation KPStarter waits 1–2 seconds.

RandomKP is a KP that randomly interacts with the SIB. The different operations are query, insert, remove, subscribe and unsubscribe. In its main loop, which runs in its own thread, it makes a maximum of one of these operations and waits 1–5 seconds.

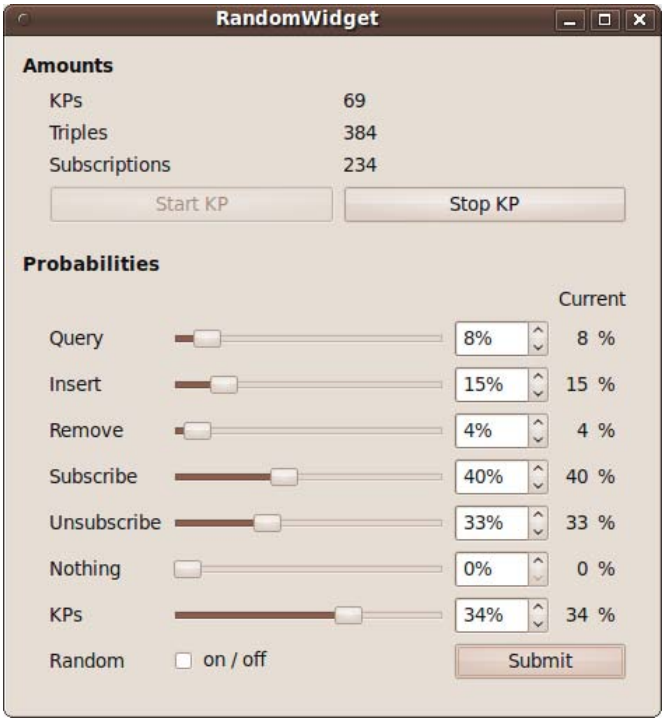


Figure 10. The RandomWidget UI.

The probabilities of different operations can be changed using corresponding UI sliders or clicking on the 'Random' checkbox which causes every RandomKP instance to create random probabilities for every operation. The 'Nothing' probability is the difference between the sum of other operation probabilities and 100%. If the operation is insert, remove, subscribe or unsubscribe, RandomKP notifies KPStarter of the parameter value change. KPStarter then updates the new value to the SIB and sends a signal to UI thread to update the UI (this is not shown in the activity diagram).

The average rate of incoming operations to the SIB can be calculated as follows:

$$r = \left(\frac{\sum_{i \in O} p(i)}{3} \right) N \tag{15}$$

where O is the set of five operations, $p(i)$ is the probability of operation i and N is the number of RandomKPs. The number 3 in the denominator is the average delay between probability evaluations. In the equation all KPs are supposed to use the same probabilities. The rate of one operation can be calculated by substituting the summation with the desired probability. If the KP creating probability is constant and positive, the value of N in time t can be calculated as:

$$N(t) = \left(\frac{p_k}{1.5} \right) t \tag{16}$$

where p_k is the KP creating probability and t is time in seconds. 1.5 is the average delay between KP creation probability evaluations. Thus Equation 15 can be expressed as:

$$r(t) = \left(\frac{\sum_{i \in O} p(i)}{3} \right) \left(\frac{p_k}{1.5} \right) t \tag{17}$$

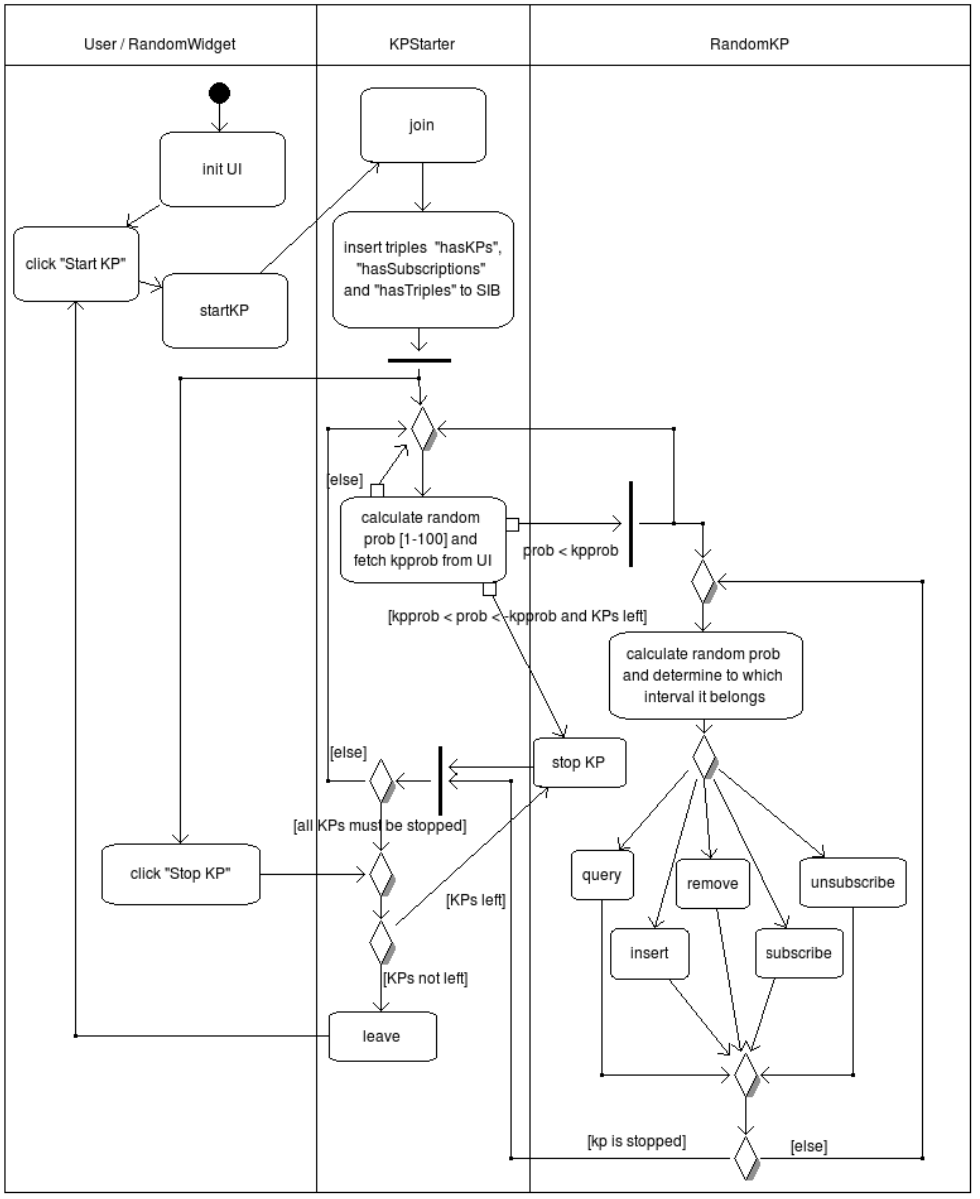


Figure 11. An activity diagram of RandomKP.

which shows that the incoming operation rate increases steadily over time if p_k and $p(i)$ are constants. This equation assumes that in the case of the remove and unsubscribe operations there are always triples to remove or subscriptions to close. However, this is not a correct assumption if the probability of remove or unsubscribe is greater than that of insert or subscribe, respectively.

PredictorKP

The most important classes of PredictorKP are PredictorWidget, MeasurerKP and Predictor and its sub-classes. PredictorWidget is the UI that shows the current status of the application. MeasurerKP makes queries to the SIB and measures the times it takes to receive responses. Predictor is an interface class with virtual functions `predict()`

and `train()`. The sub-classes of Predictor are used to make predictions for the latency measurement. The class diagram of PredictorKP is shown in Figure 12. The KPInterface class is the same as the one used in RandomKP.

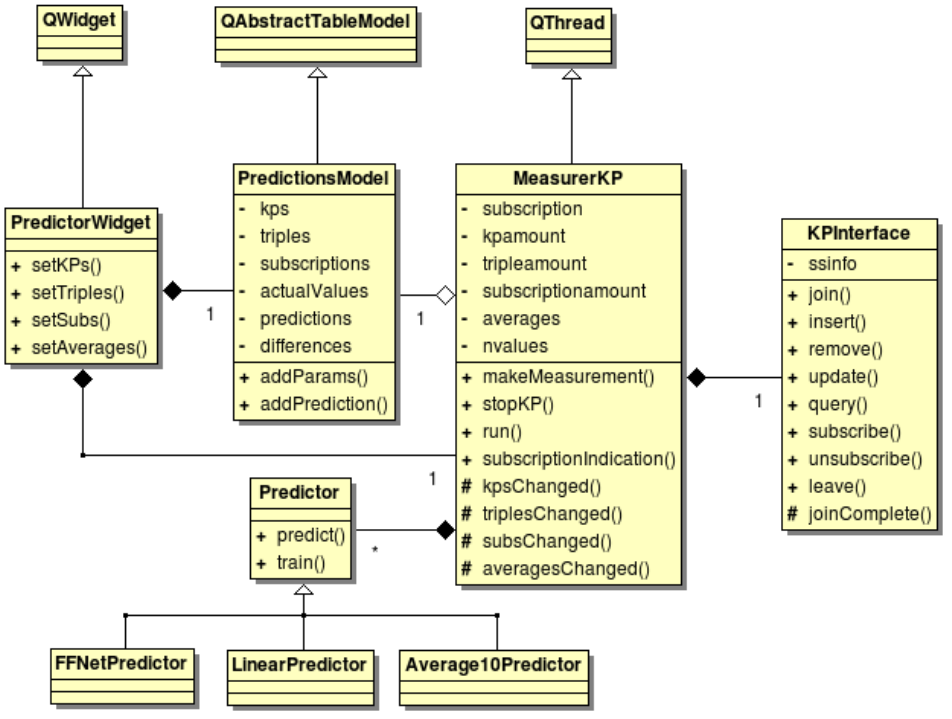


Figure 12. The PredictorKP class diagram.

The PredictorWidget UI is shown in Figure 13. The bottom-left corner of the window shows the current values of the used parameters. The 'Start KP' button creates a MeasurerKP instance and the 'Stop KP' button deletes it. The table in the right side of the window shows the measurements that have been made. The first three columns show the values of the predictions parameters at the time of measuring and the fourth column shows the actual duration of the query. The next columns show the predictions of different predictors along with the difference between the prediction and the actual value.

MeasurerKP is used to handle the Predictor instances and to make and handle measurements. When started, it joins to the SIB, subscribes to the KP, triple and subscription amounts and inserts ten triples which have the same subject and predicate but different objects. These triples are then queried in ten seconds intervals using a template query with the object element as a wildcard. When the query is made, different Predictor instances are used to estimate the latency. The numbers of KPs, subscriptions and triples are given to the predictors as parameters. After that, the predictors are trained using the actual latency value. The predictions and other values are updated to the table in the UI and written to log files.

The sub-classes of the Predictor class estimate the time to make the query when the SIB has the given parameters. The `predict()` function is used to create the estimate based on the parameters and the internal model of the instance and the `train()`

function is used to refine the internal model using the actual desired output for the parameters. In this case three different predictors were implemented and used: LinearPredictor, Average10Predictor and FFNetPredictor. Although each could be considered to be machine learning agent by the definition that they improve their function by experience, Linear and Average10 are very simple models and they are used as a reference for testing the neural network implementation FFNet.

LinearPredictor calculates the estimation based on the previous measurements using the linear function:

$$p_{t+1} = ((100 - w)p_t + wm)/100 \tag{18}$$

where p_t is the prediction at iteration t. The variable m is the latest measured value and w is its weight. The weight starts from 100 but it decreases to 10 so that later the previous measurements have a greater impact on the prediction. Thus LinearPredictor can be expected to adapt quite slowly to changes in the latencies. LinearPredictor does not use the triple, KP and subscription amounts to create the prediction.

Average10Predictor is another reference predictor that behaves quite similarly to LinearPredictor. It stores ten previous measured latency values and returns the arithmetic average value of them as a prediction. Compared to LinearPredictor AveragePredictor should react a bit more rapidly to changes in measurements because it only uses ten previous measurements while all measured latencies affect the prediction of LinearPredictor.

FFNetPredictor uses a feed-forward artificial neural network to create the prediction and here it used the neural network implementation of the Shark machine learning library [63] was used. The network has three input units, three hidden units and one output which is the latency estimate. The hidden units are sigmoid units, but because they produce results between 0 and 1, the use of them in duration evaluation would require scaling. Therefore the output unit is a linear unit in which the output is a weighted sum of the inputs. Thus the output can be directly used as a duration estimate.

The learning algorithm used in the experiment was the IRPropPlus (improved Resilient-Backpropagation-algorithm with weight-backtracking [64]) algorithm from Shark which is based on the backpropagation algorithm mentioned in Section 3.5.4. When used, the network is trained continuously after each measurement. Thirty previ-

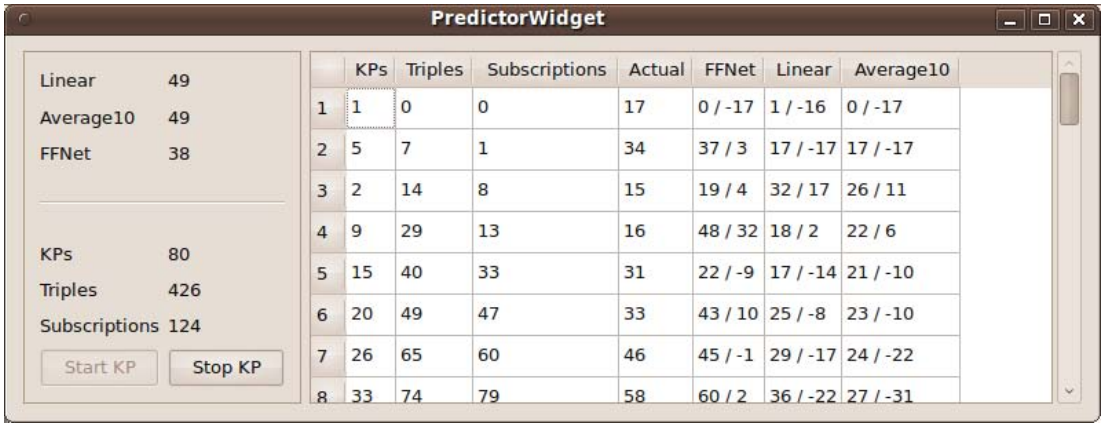


Figure 13. The PredictorWidget UI.

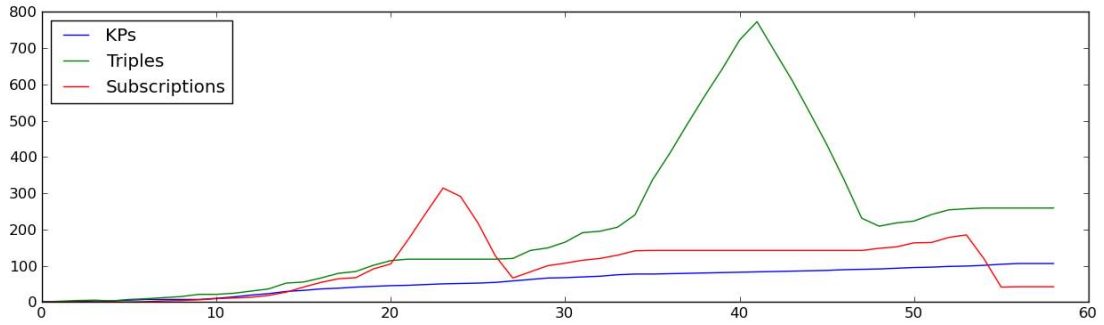
ous measurements are saved and used in training. The training algorithm makes fifty iterations with the thirty training inputs and outputs.

5.1.2. Evaluation

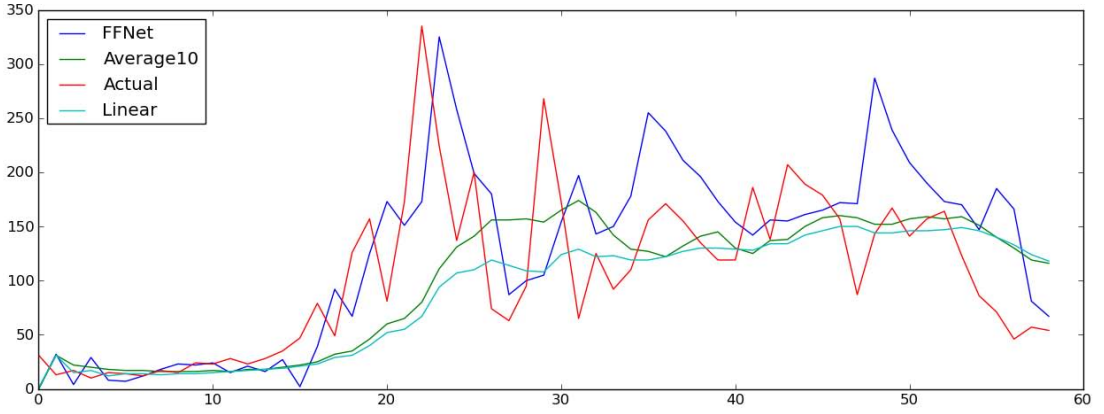
The applicability of machine learning algorithms to the case was evaluated using the KPs described in the previous section. The Python SIB was run on a Dell Latitude D410 laptop[2] and RandomKP and PredictorKP applications were run on a Dell Latitude D830 laptop[3]. Both machines were in the same local network. The applications were run for about ten minutes and in that time the MeasurerKP made a total of 59 measurements. The three predictors described in the previous section were used to estimate the latency of each query and then the measurement was used to train them. The parameter values of the different measurements are shown in Figure 14a and the measured and estimated latencies are shown in Figure 14b.

As can be deduced from the figures, random probabilities for RandomKP instances were used for approximately the first 200 seconds (20 measurements). Only the KP

[2]http://www.dell.com/downloads/global/products/latit/en/spec_latit_d410_en.pdf
[3]http://www.dell.com/downloads/global/products/latit/en/spec_latit_d830_en.pdf



(a) Parameter values.



(b) Latencies and predictions.

Figure 14. The parameter values, measured latencies and latency estimates of the test run.

amount increases quite rapidly. It should be noted that it is logical that both subscription and triple insert amounts are almost always greater than KP values because the KPs do not remove the triples that other KPs have inserted and cannot unsubscribe other KPs' subscriptions. Therefore the triples that have the greater probability for the remove and/or unsubscribe operations than for insert/subscribe do not affect the overall behaviour too much. In this phase all the used predictors seem to perform quite equally.

When the probability, and consequently number of subscriptions, is increased the measured latency also increases dramatically. As expected, the reference predictors do not react to this change very quickly but the FFNetPredictor adapts much better to this change. However, the zigzag curve seen in the figure is very characteristic to the measured latencies and there seems to be no correspondence between this and the used parameter values. Therefore FFNetPredictor is often unable to predict the latencies and often seems to be a bit 'late'.

Later when the amount of inserted triples makes a brief peak it can be seen that the change (at least a change of this magnitude) does not substantially affect the latency. However, the FFNetPredictor also makes a short peak when the amount of triples begins to increase. It shows that the predictor had made a false generalisation that the triple amount would affect the latency more but it was able to correct this erroneous assumption.

The impact of the KP amount was not tested. The expected impact is not direct and comes from the amount of traffic and processing they require. However, different KPs behave differently and it is difficult to find any pattern. Notably, in this test the behaviour of the KPs changes over time. However, when speaking about real environments using Smart-M3 IOP, there may be thousands of KPs joined to an SIB. In that case it might be possible that a pattern could be found in the relationship between the KP amount and latency.

5.2. Decision Making

The previous case was about latency prediction which helps in making adaptation decisions. However, raw latency information is often not enough to find the optimal decisions because of the possible trade-off between different aspects of quality. Reinforcement learning can be used to try out different decisions in different states and thus learn how to achieve maximum utility or reward.

In this case reinforcement learning is used to find a good policy for making decisions. The problem is to find the optimal resource allocation between different devices. A distributed video-streaming application was chosen as an example. It consists of a video-capturing component, a colour-space converter component and a video-display component. In this case every device in the smart space runs every component type. When the source (capture) and sink (display) components are chosen the learning agent must try to choose the optimal filter (colour-space converter) to get the maximum reward. The environment calculates the reward that is given to the agent using the time to establish the connections between the components and the processor utilisation rates of the devices as parameters. However, the learning agent does not know this function

and it must try to estimate it using only the processor performance information from different devices.

As can be seen, there is a possible a trade-off between latency and load distribution. If every component is chosen from a different device the latency will probably be large because of network latencies. However, it distributes the processing load very efficiently. Running all components in one device may increase the processor utilisation to an unacceptably high rate although it minimises the network latency.

5.2.1. Implementation

This case was implemented using four different KPs, some of which have multiple instances running (a short name is in parentheses):

1. A KP that provides a single multi-media filtering service (PyGSTKP).
2. A manager KP that inserts device information into the SIB and is used to start and stop PyGSTKP KPs in a device (Manager).
3. A KP that chooses the source and sink elements of the video streamer and gives rewards (Task).
4. A learner KP that chooses a filter element and receives rewards for the task (Learner).

The cooperation of these KPs is illustrated as a sequence diagram in Figure 15. The KPs are described in more detail in the following sections. The descriptions focus more on the internal functionality of the KP applications but they also clarify this information-level cooperation so it is advisable to occasionally refer to this figure while reading.

PyGSTKP

PyGSTKP is a KP that uses the GStreamer multi-media framework[4] and its Python bindings[5] for multi-media processing. To better understand the operation of PyGSTKP an explanation of the GStreamer framework is necessary.

GStreamer is a framework that allows the creation of any type of streaming multi-media applications. It is based on plug-ins that provide the needed functionality and is thus very extensible. There are some basic plug-ins in the core installation of GStreamer and it is possible to install plug-in packages or create your own plug-ins for needed tasks. The plug-ins can be classified into six groups: protocol handling, sources, formats, codecs, filters and sinks. GStreamer is written in C language and its type system is based on the GObject library. [65]

The most important types defined in GStreamer are Element, Pad and Pipeline. Element performs one specific function such as reading, decoding or outputting data. Elements have input (sink) and output (source) pads that can be connected to other

[4] http://gstreamer.freedesktop.org/

[5] http://pygstdocs.berlios.de/

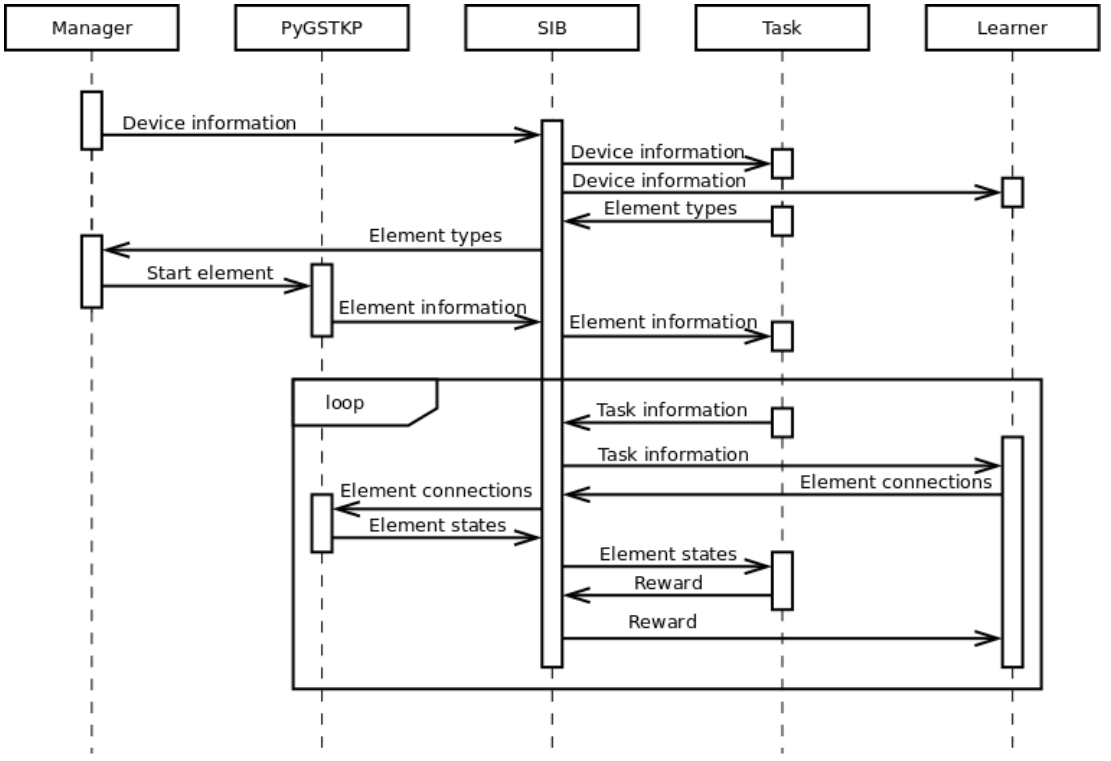


Figure 15. Sequence diagram of the decision making test case.

elements' pads. Every pad is associated with a capability and connected pads' capabilities must be compatible. Capability describes the media type that the pad can handle or output. Connected elements create a pipeline which performs a certain task, for example media playback. Pipelines are sub-classes of elements which reduce the complexity of GStreamer a little. Elements (and pipelines) can be in one of four states: NULL, READY, PAUSED or PLAYING. [65]

Figure 16 shows a GStreamer pipeline of an Ogg player that supports Vorbis and Theora codecs. The figure is adapted from the GStreamer Application Development Manual [65]. The pipeline consists of six elements: a source, a demuxer, two decoders and two sinks. As can be seen in the figure, source elements do not have sink pads and sink elements do not have source pads. Demuxers create a source pad for every stream type contained in the incoming stream.

PyGSTKP KP is a wrapper around one GStreamer element. It is used to receive a stream over a UDP (User Datagram Protocol) link, stream it through the defined element and send it again over a UDP link to another PyGSTKP KP. It creates a pipeline based on the one shown in Figure 17 (pads are not explicitly shown) and sends information about it to the SIB. The third element in the pipeline, labelled <*element*>, is the element that is wrapped and its type is given as a command-line argument. The other elements are included in GStreamer plugin packages. The *udpsrc* element is used to receive a stream coming to a specified port and deliver it to the pipeline. The *capsfilter* elements are used to restrict the media type of the stream. It must be used after the udpsrc element because udpsrc cannot determine the media type and thus capability negotiation in the pipeline would not be possible. The capsfilter after the main element

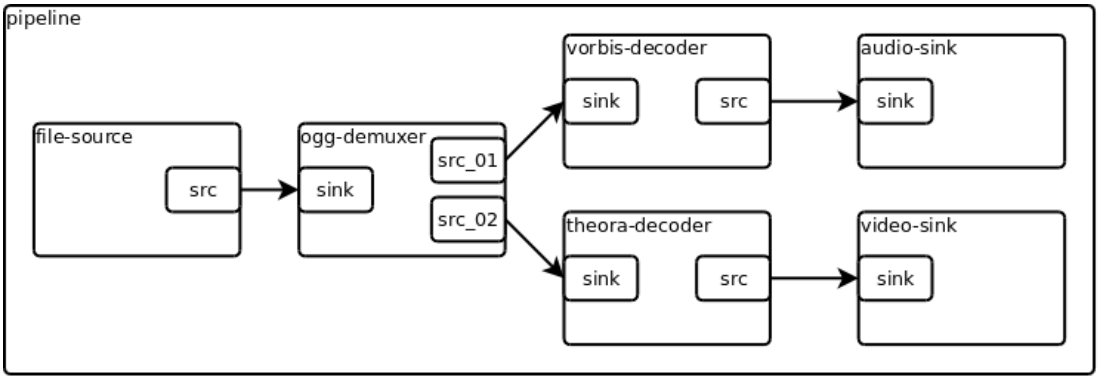


Figure 16. GStreamer pipeline for a basic Ogg player.

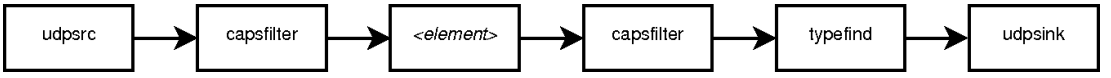


Figure 17. Base model of the pipelines used in PyGSTKP.

is present only if the desired capabilities of the element are given as command-line arguments. The *typefind* element is used to find the media type of the stream: this media type is announced in the SIB. The *udpsink* element sends the stream to the next PyGSTKP KP. In fact, the udpsink element is only used when actually streaming. Otherwise it is substituted with a *fakesink* element which just discards the incoming stream. When the element is a source element the udpsrc element and the first capsfilter element are not present in the pipeline. Similarly, in the case of a sink element, the last three elements are not present.

There is an obvious problem in this structure when an element output is in a raw format. Usually in cases with compressed media formats there are marks where decoding units start and end. However, for example, in the case of raw video, it is not possible to know where video frames start and end because there are no start and end marks. A workaround was created for this demonstration. If the media format of an element is raw video (*video/x-raw-rgb* or *video/x-raw-yuv*) it is encoded to an MJPEG (Motion JPEG) stream where every video frame is separately encoded to a JPEG image with certain repetitive fields omitted. This solution adds a computation load to elements that produce raw video and decreases network bandwidth need. However, since the goal of the implementation was to demonstrate reinforcement-learning capabilities and not to create a high-performance video decoder, this approach is suitable.

PyGSTKP inserts information about itself according to an ontology defined for this purpose into the SIB. Figure 18 shows an example RDF sub-graph created according to the ontology. This graph is created by a KP wrapping an *ffmpegcolorspace* element which is used to make colour-space conversions. In the graph the nodes beginning with the character '_' represent blank nodes and they are of the type indicated by the name, for example the node *_device* is of type *gs:Device*. These type triples are not shown in the figure to keep it simple. All self-defined URIs in this case are in the name-space *gs*. The ffmpegcolorspace element is of type *gs:Filter* which means that it can have both source and sink pads. Other element types are *gs:Source* which has only

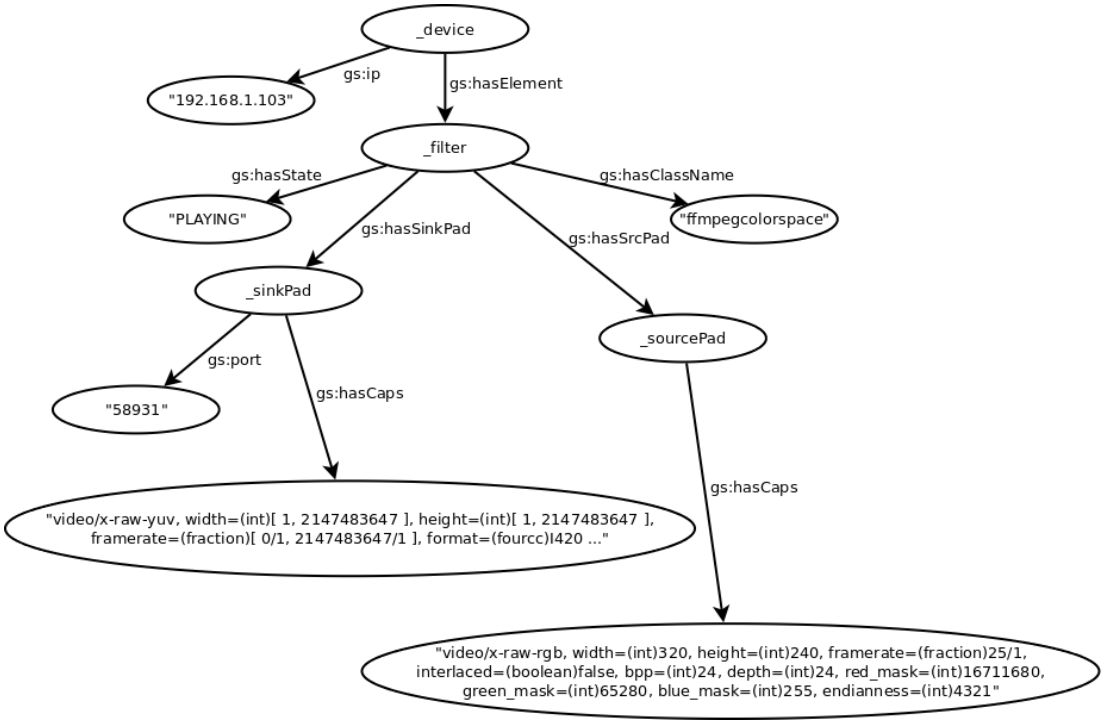


Figure 18. An RDF subgraph created by a PyGSTKP KP wrapping an *ffmpeg-colorspace* filter.

source pads and *gs:Sink* which has only sink pads. The sink-pad information of an element is always inserted to the SIB but source-pad information is only inserted when the element is ready to send the stream to another KP. For source elements the source-pad information is thus always inserted, but for filter elements only when their sink pads are connected to other elements. Both *gs:SinkPad* and *gs:SourcePad* elements are associated with capabilities. However, the sink pads' capabilities are not fixed and they show all the possible media formats that the element can handle. For example in the case of the ffmpegcolorspace element there are over thirty possible formats (only the first is shown in the graph to save space). In case of source pads, the capabilities are fixed and they tell the exact media format that the element is streaming. Every sink pad is listening to a port to which other elements can send media stream. The *gs:hasState* property tells the state of the pipeline of the KP.

The connections between elements are created by an external KP. It inserts a triple connecting a source pad and a sink pad with the predicate *gs:connectedTo*. PyGSTKP KPs subscribe to these kind of triples for both their source and sink pads. When a connection is made to a sink pad, the sink PyGSTKP queries the announced capabilities of the sending source pad from the SIB and adjusts the capsfilter element accordingly. Then it starts the pipeline and, when the typefind element finds the media type of the source pad, it inserts the source pad information into the SIB. When a connection is made to a source pad, PyGSTKP asks for the device's IP address and the sink pad UDP port of the receiver element and starts sending stream to that UDP socket. So a KP can create 'pipelines' consisting of PyGSTKP elements by creating connections between pads in the SIB.

Manager

Manager is a simple KP whose main tasks are starting (and stopping) PyGSTKP processes and inserting device information into the SIB. The device information contains the IP address, processor performance and processor usage of the device. The IP address and processor performance are inserted into the SIB only when the Manager KP joins it but the processor usage is updated every two seconds. The processor performance is estimated using the `bogomips` value in Linux virtual file `/proc/cpuinfo`. Although it is a very poor measure for processor performance [66], it is applicable in this case because it manages to divide the devices used in the demonstration into three categories. If the `bogomips` value is less than 800, the processor performance is 'Low'. If the value is more than 800 but less than 3000, the performance is 'Medium'. If it is more than 3000, the performance is 'High'. The average processor usage rate is calculated for the last two seconds using the values read from the Linux virtual file `/proc/stat`.

After inserting the device information into the SIB the Manager KP subscribes to the element type and reset triples with the following formats: *(_device, gs:hasGSTKP, x)* and *(_device, gs:reset, x)* where *x*, the object of the triple, is not defined in the subscriptions. When a triple, with the predicate *gs:hasGSTKP*, is inserted the Manager starts a PyGSTKP sub-process with the arguments given as the triple object. When a triple, with the predicate *gs:reset*, is inserted the Manager kills all PyGSTKP processes and starts them again. This feature is actually not needed in this demonstration but is added to increase the usability of the KP.

Task

The Task KP has two main duties. It allows the user to control the types of PyGSTKP KPs that the different devices run and it also handles the setting of the service discovery tasks and rewards. The user interface of the application is shown in Figure 19.

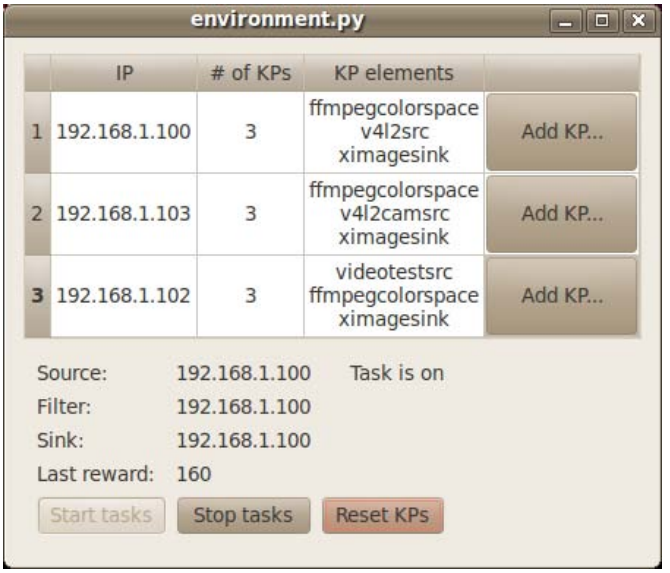


Figure 19. The user interface of the Environment KP.

The Task KP gets subscribe indications whenever a new device joins the smart space. It adds the IP address of the device to the first column of the table in the UI. It also makes subscriptions to the elements of the device (see the RDF graph in Figure 18). It shows the number of elements in a device and the element class names (found from the triple with the predicate *gs:hasClassName*, see Figure 18) in the next columns. It also saves the element URIs and uses them in task generation. By clicking on the 'Add KP...' button the user receives a dialogue box where the arguments of a new PyGSTKP process can be typed. That causes the device manager to start a new PyGSTKP process and subsequently the element information to be shown in the UI. By clicking on the 'Reset KPs' button the Task KP sends a reset triple to every device and this causes the device managers to restart the PyGSTKP processes.

The 'Start tasks' button causes the Task KP to start a task simulation. It randomly chooses one source and one sink element URI and inserts them into the SIB as shown in Figure 20. The learning agent must then choose the optimal filter between the source and sink to make the streaming between the source and sink possible. The Task KP subscribes to the state of the sink element: when it is PLAYING, it knows that correct connections have been made. Then it inserts a reward to the task (also shown in Figure 20). The reward is calculated using the following equation:

$$r = 200 - \bar{u} - s_n - 10t \tag{19}$$

where r is the reward, $\bar{u}$ is the average processor utilisation of all the devices, s_n is the standard deviation of the processor utilisation rates and t is the duration of the task in seconds. The average processor utilisation is calculated using the following equation:

$$\bar{u} = \frac{1}{N}\sum_{i=1}^{N} u_i \tag{20}$$

where N is the number of devices and u_i is the processor utilisation rate of the device i. The standard deviation is calculated using the following equation:

$$s_n = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(u_i - \bar{u})^2} \tag{21}$$

and it is included because the utilisation rates are required to be as even as possible. If the learning agent has not chosen a filter in ten seconds, a reward of 0 is inserted and the task is cancelled.

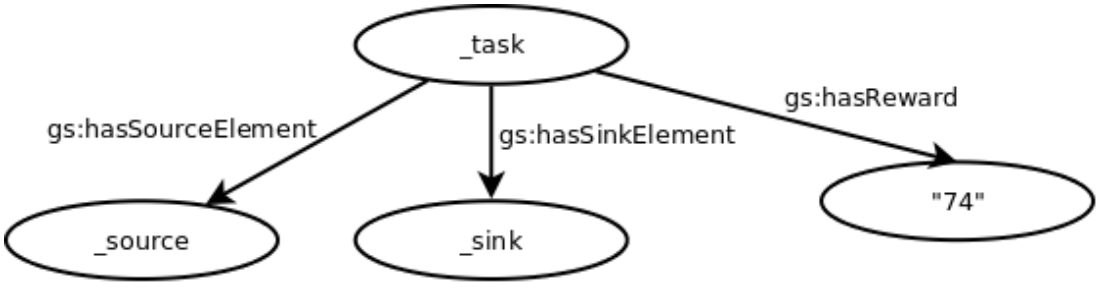


Figure 20. An RDF subgraph created by a Task KP describing a task.

After the reward is inserted, the Task KP removes all the connections made by the learning agent, waits a few seconds for the PyGSTKP instances to revert to the start state and inserts a new task with a new source and sink. It continues this until the user stops the loop by clicking on the 'Stop tasks' button.

Learner

The Learner KP chooses a filter for a task the Task KP has inserted. It uses reinforcement learning with a look-up table to estimate the rewards for state–action pairs. The KP uses base classes defined in the PyBrain library [67] and extends some of them to overcome the challenges caused by asynchronous communication and distributed environment.

The look-up table of Learner has entries for nine states and five actions: 45 cells in total. The states are determined by the computing performances of the devices in which the source and sink elements are as reported by the Manager KPs. Because there are three possible performance levels, there are nine different permutations of source–sink performance pairs which are regarded different states. The actions correspond to the filters which are chosen to complete the pipeline. Because of the assumption that network latencies and unevenness in resource usage cause worse results, the possible actions are chosen so that they have an effect to these issues. The five actions are abbreviated as SAME1, SAME2, DIFFLO, DIFFME and DIFFHI (meaning the same device as the source, the same device as the sink, a different device from the source and sink with low performance, a different device with medium performance and a different device with high performance, respectively). If both the source and the sink element are in the same device, SAME1 and SAME2 are essentially the same actions. If there are less than three devices belonging one of the performance categories there will be states in which some of the last three actions are impossible. For example, if there is only one device with medium performance and it is the same as the source in a task it is impossible to choose a device that is different from source and has medium performance.

The look-up table is initialised with values of 1000 to ensure that every action will be explored in a reasonable time. It uses the Q learning algorithm to find estimates for rewards. The Q learning training rule was presented in Equation 14 and is repeated here:

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')]. \tag{22}$$

Because the problem does not include state transitions, the equation can be somewhat simplified. The variable γ, the weight of future rewards, can be thought to be zero and the Q value reduces in direct relation to the expected reward for a state–action pair. The simplified training rule is:

$$\hat{r}_n(s,a) \leftarrow (1-\alpha_n)\hat{r}_{n-1}(s,a) + \alpha_n r \tag{23}$$

where $\hat{r}_n(s,a)$ is the Learner's estimate for the reward in time n. The learning rate in time n, α_n, is calculated as follows:

$$\alpha_n = \begin{cases} \frac{1}{k(s,a)} & \text{if } k(s,a) \leq 10 \\ 0.1 & \text{if } k(s,a) > 10 \end{cases} \tag{24}$$

where $k(s, a)$ is the number of times that action a has been taken in state s. The minimum value for the learning rate is 0.1. For the first ten iterations of a state–action pair the training rule calculates the arithmetic mean value of all training examples but after that the last value always has an impact of ten per cent of the $\hat{r}$ value.

To handle the trade-off between exploration and exploitation Learner uses PyBrain's implementation of the epsilon-greedy exploration strategy. It has two parameters: epsilon (ϵ) which determines the possibility of exploring and decay which determines the decreasing speed of the epsilon parameter. So when an action is selected, there is a possibility of $1 - \epsilon$ to choose the action that currently seems to be the best and a possibility of ϵ to randomly choose any of the available actions. After every action the epsilon parameter is decreased by multiplying it with the decay parameter. In the used implementation, the epsilon parameter was initialised to 0.3 and the decay parameter was 0.9999. Thus the exploration probability was quite high and decreased slowly. It takes over 34,000 iterations to decrease to below 0.01.

When an action is chosen the Learner waits for the Task to insert a reward for the task. This reward is then used for the training. After that, the Learner waits for a new task. However, when the Learner chooses an action that is impossible, a reward of -100 is used for training and a new action is chosen for the same task: in this way impossible actions are not inserted into the SIB.

5.2.2. Evaluation

The case was evaluated using test runs of different lengths. In this section one of the test runs is described. The equipment used in the test was a Dell Latitude D830 laptop[6], a Dell Latitude D410 laptop[7], a Nokia N900 smart phone[8], a Linksys WRT54GL router[9] and a Logitech QuickCam Deluxe for Notebooks web camera. The Latitude D410 laptop had Ubuntu Linux 10.04 natively installed and the Latitude D830 laptop ran two virtual machines with Ubuntu operating system versions 9.10 and 10.04 using VirtualBox virtualisation software[10] with a Microsoft Windows XP operating system. All the devices were connected using the WRT54GL router, N900 over Wi-Fi and the others using Ethernet cables. The virtual machines used bridged virtual network adapters to get IP addresses in the same address space as the physical devices. The web camera was connected to the Dell Latitude D410 laptop. Figure 21 shows the composition of the hardware equipment and software in the test.

The SIB was located in the virtual Ubuntu 9.10 machine. The other Ubuntu machines and Nokia N900 all ran Manager KPs. The virtual Ubuntu 10.04 ran also the Task and Learner KPs. Using the Task KP, the needed PyGSTKP KPs were started on all three machines. Every device had a sink KP with an *ximagesink* GStreamer element and a filter KP with a *ffmpegcolorspace* element which changed the media types of the streams from *video/x-raw-yuv* to *video/x-raw-rgb*. In the source KP the virtual Ubuntu machine had a *videotestsrc* element, the native Ubuntu KP had a *v4l2src* element and

[6] http://www.dell.com/downloads/global/products/latit/en/spec_latit_d830_en.pdf
[7] http://www.dell.com/downloads/global/products/latit/en/spec_latit_d410_en.pdf
[8] http://europe.nokia.com/find-products/devices/nokia-n900/specifications
[9] http://www.linksysbycisco.com/EU/en/products/WRT54GL
[10] http://www.virtualbox.org/

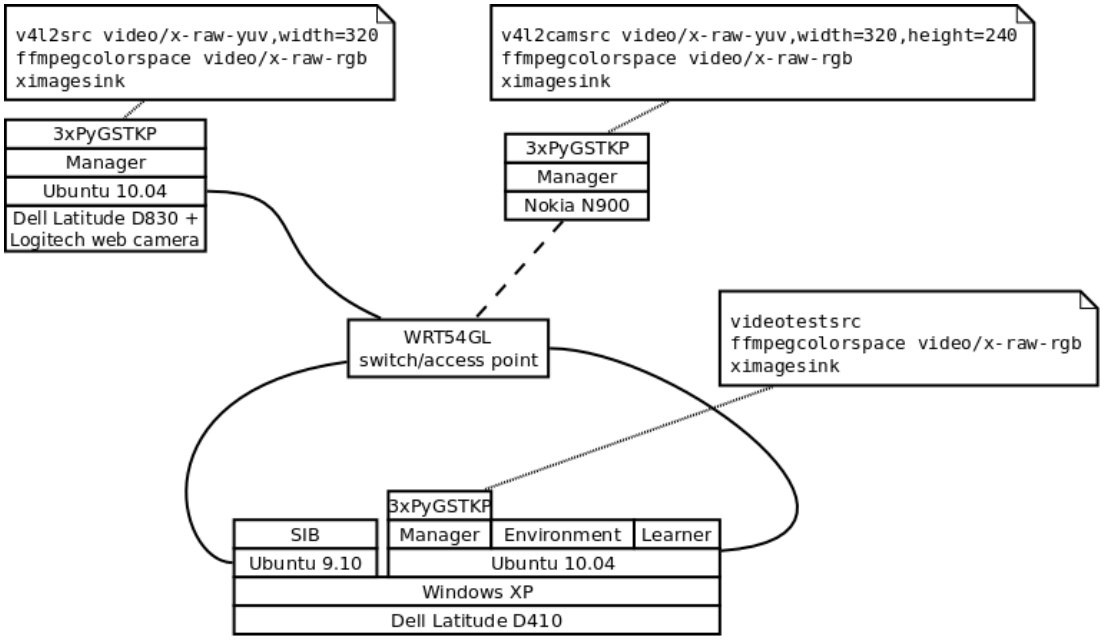


Figure 21. The composition of hardware and software in the decision making tests.

the N900 had a *v4l2camsrc* element. The reason that the source elements were different is that the machine with *videotestsrc* has no video camera and thus used test video and the needed output format (*video/x-raw-yuv,width=320,height=240*) is supported by *v4l2camsrc* in N900 in contrast to *v4l2src*. All video sources were captured using this format. The *v4l2src* element works well with the Ubuntu laptop and the webcam.

The test was run overnight and it lasted for 982 minutes (sixteen hours and twenty minutes). During this time the system made 6212 experiments, 390 of which were impossible actions chosen by the learner and 5722 were actual decisions. Thus the system made an average of 5.8 experiments per minute.

The look-up table of the learner used in the experiment is shown in Table 5. The rows show the states of the environment as seen by the decision maker. They are shown as (<*source device performance*> / <*sink device performance*>) pairs where LO, ME and HI mean low, medium and high performance, respectively. The columns show the action which is the chosen filter device. SAME1 means the same device as the source, SAME2 is the same device as the sink and DIFF<x> means a different device where <x> is LO, ME or HI, meaning same things as previously mentioned. The cells show the estimates of the rewards for all the possible state–action pairs. The rewards of impossible actions are shown as a dash. The numbers of times the learner tried an action in a state are shown in parentheses.

As can be seen in the table, the learner chose the action with the biggest expected reward most of the time. However, for example, in the third state, *LO/HI*, there are two actions with expected rewards that are very close to each other. In this situation the decision maker changed its behaviour according to its view of the environment at that time. Similar behaviour can be seen, for example, in the last state, *HI/HI*, although not as strongly.

Table 5. The look-up table of the decision maker after 6212 experiments.

	1. SAME1	2. SAME2	3. DIFFLO	4. DIFFME	5. DIFFHI
1. LO/LO	0.17 (52)	0.16 (53)	- (32)	0.14 (148)	0.15 (367)
2. LO/ME	3.29 (31)	124.35 (124)	- (33)	- (28)	140.37 (483)
3. LO/HI	0.66 (38)	124.51 (397)	- (25)	125.77 (209)	- (34)
4. ME/LO	0.07 (159)	0.07 (60)	- (39)	- (20)	0.07 (413)
5. ME/ME	137.97 (41)	139.84 (33)	3.53 (18)	- (28)	151.14 (563)
6. ME/HI	147.39 (37)	151.67 (568)	2.27 (29)	- (46)	- (33)
7. HI/LO	0.04 (484)	0.04 (69)	- (44)	0.04 (102)	- (32)
8. HI/ME	165.35 (554)	154.24 (48)	2.99 (35)	- (38)	- (32)
9. HI/HI	163.23 (127)	164.24 (363)	6.32 (15)	155.07 (101)	- (26)

All the state–action pairs in the table that include the Nokia N900 (device with low performance) as a filter or sink are near zero in the table. This is erroneous behaviour caused by the KPs on the N900 going to an unresponsive state. The exact reason for this fault is not known and it seems to occur only with the N900. In this experiment the filter KP failed after 372 experiments and the sink KP failed after 4446 experiments. Table 6 and Table 7 show the look-up tables after 372 and 4446 experiments, respectively. Although some state–action pairs are visited only a few times, especially in Table 6, the expected rewards are usually of the same magnitude as in Table 5. Therefore it is also feasible to make an evaluation based on these tables. Table 8 shows a combination of Tables 5, 6 and 7 where the erroneous values have been substituted with values from earlier tables. The values with a bold font are from Table 6 and the values with an italic font are from Table 7, other values are from Table 5. Table 9 shows a simplified view of Table 8. The table shows the expected rewards for the selections of different devices. If there are two entries for the same state–action pair the larger one was chosen.

Table 6. The look-up table of the decision maker after 372 experiments.

	1. SAME1	2. SAME2	3. DIFFLO	4. DIFFME	5. DIFFHI
1. LO/LO	69.00 (2)	74.00 (2)	- (2)	107.03 (17)	88.00 (3)
2. LO/ME	75.25 (4)	123.83 (6)	- (7)	- (2)	137.96 (26)
3. LO/HI	63.50 (2)	149.67 (25)	- (4)	137.00 (8)	- (4)
4. ME/LO	97.29 (28)	67.50 (2)	- (4)	- (3)	91.13 (8)
5. ME/ME	130.67 (3)	138.00 (1)	82.00 (1)	- (3)	145.84 (33)
6. ME/HI	145.14 (7)	158.51 (30)	84.00 (2)	- (2)	- (3)
7. HI/LO	106.61 (25)	68.67 (3)	- (4)	85.67 (3)	- (2)
8. HI/ME	170.68 (32)	155.60 (5)	104.25 (4)	- (4)	- (3)
9. HI/HI	154.00 (1)	166.63 (25)	53.50 (2)	156.45 (11)	- (4)

Table 7. The look-up table of the decision maker after 4446 experiments.

	1. SAME1	**2. SAME2**	**3. DIFFLO**	**4. DIFFME**	**5. DIFFHI**
1. LO/LO	4.81 (20)	6.37 (18)	- (22)	94.95 (86)	103.01 (305)
2. LO/ME	8.50 (22)	120.06 (117)	- (28)	- (26)	136.36 (331)
3. LO/HI	1.91 (28)	143.91 (339)	- (22)	131.03 (73)	- (30)
4. ME/LO	92.52 (91)	3.09 (24)	- (27)	- (17)	119.11 (343)
5. ME/ME	139.00 (34)	137.27 (24)	6.64 (12)	- (20)	154.43 (392)
6. ME/HI	145.97 (29)	152.08 (408)	5.27 (21)	- (30)	- (20)
7. HI/LO	117.59 (409)	9.85 (17)	- (29)	103.31 (28)	- (24)
8. HI/ME	161.73 (379)	151.69 (38)	3.70 (33)	- (30)	- (27)
9. HI/HI	168.67 (47)	162.73 (297)	9.63 (11)	157.68 (97)	- (21)

Table 8. A combination of the look-up tables of the decision maker.

	1. SAME1	**2. SAME2**	**3. DIFFLO**	**4. DIFFME**	**5. DIFFHI**
1. LO/LO	**69.00 (2)**	**74.00 (2)**	- (32)	*94.95 (86)*	*103.01 (305)*
2. LO/ME	**75.25 (4)**	124.35 (124)	- (33)	- (28)	140.37 (483)
3. LO/HI	**63.50 (2)**	124.51 (397)	- (25)	125.77 (209)	- (34)
4. ME/LO	*92.52 (91)*	**67.50 (2)**	- (39)	- (20)	*119.11 (343)*
5. ME/ME	137.97 (41)	139.84 (33)	**82.00 (1)**	- (28)	151.14 (563)
6. ME/HI	147.39 (37)	151.67 (568)	**84.00 (2)**	- (46)	- (33)
7. HI/LO	*117.59 (409)*	**68.67 (3)**	- (44)	*103.31 (28)*	- (32)
8. HI/ME	165.35 (554)	154.24 (48)	**104.25 (4)**	- (38)	- (32)
9. HI/HI	163.23 (127)	164.24 (363)	**53.50 (2)**	155.07 (101)	- (26)

Table 9. A simplified view of Table 8.

	LOW	**MEDIUM**	**HIGH**
1. LO/LO	**74.00**	*94.95*	*103.01*
2. LO/ME	**75.25**	124.35	140.37
3. LO/HI	**63.50**	125.77	124.51
4. ME/LO	**67.50**	*92.52*	*119.11*
5. ME/ME	**82.00**	139.84	151.14
6. ME/HI	**84.00**	147.39	151.67
7. HI/LO	**68.67**	*103.31*	*117.59*
8. HI/ME	**104.25**	154.24	165.35
9. HI/HI	**53.50**	155.07	164.24

From the tables it can be seen that it was not the best decision to use the low-performance device as a filter. The values of such cells (written in bold font) are remarkably lower than the other rewards for the states. In fact, in all but one state the action giving the best reward was to choose the high-performance device as a filter. In state number 3, LO/HI, the medium-performance device seems to give a slightly better reward. However, the difference between the rewards is not significant and as can be seen from the times that the filters were chosen, action number 2 – high performance device – was used more than the medium one and thus had the biggest reward value most of the time.

There is also another situation in which substituting a component with another with better performance seems to cause a worse reward. Compared to pipeline medium–high–low, the reward of pipeline high–high–low is inferior. In addition to that, the reward of pipeline low–medium–medium is very close to the rewards of the aforementioned pipelines low–medium–high and low–high–high. In these situations there is no clear advantage in using high-performance components.

6. Discussion

This chapter analyses the results of the thesis and discusses the success of the implemented demonstrations. Some improvements and future work related to Smart-M3 generally, and specifically using machine learning in it, are suggested.

6.1. The Latency Prediction Case

The tests indicate that the use of machine learning techniques can help in predicting latencies. However, a really usable and adaptive latency predictor in dynamic environments is still far away. This conclusion is consistent with the results of Ipek et al. as described in Section 3.9.1.

There are many possible ways to improve the solution and results of this case. A good approach could be a predictor that is trained before it is set up for its final operating environment. This type of predictor could give reasonable predictions from the beginning and then use online learning to further improve the results and to adapt the changes in the environment. The prediction capabilities of different predictors could also be tested by using them on the same dataset.

The approach needs also more testing and validation. The evaluation presented in this thesis, and also other tests made during the implementation, give a good indication about the benefits of machine learning but more formal tests, longer test runs and bigger test environments are needed. However, the current SIB implementations are in the development phase and they contain some instabilities which makes large-scale testing very challenging. In addition, the need for bigger environments requires more equipment than it is reasonable to obtain for this kind of test.

The problems encountered while implementing this case indicate that there may be a need to improve Smart-M3 IOP and SIB implementations. Some KPs, such as the latency predictor, require reliable information about the smart space. For example, the numbers of triples, subscriptions and joined KPs or other status information could be this kind of required information. Other KPs could aldo use, for example, information about the producer and production time of the information in the SIB. These features could be implemented in the SIB and provided to all or some privileged KPs.

Additionally. the necessity for defining removed triples when updating information is a real problem in Smart-M3. In the current situation, if many KPs update the same data, the KPs must query the data before updating in order to check the current information in the SIB. However, if another KP updates the same information after the query, the first KP may use obsolete triples as removable triples in the update transaction. Because the information is no longer in the SIB, it cannot be removed but the added triple will be inserted anyway. Thus the same information will exist in two triples which is not a desired outcome. It should be made possible to make updates in which *all* triples, according to a certain pattern, are removed and new triples will be inserted.

6.2. The Decision-Making Case

The decision maker implemented in the demonstration was able to generate a model that allowed it to choose the best filter in every state. According to the test, it seems that it is possible to use reinforcement learning in decision making when it is possible to get rewards from the environment. However, in many applications getting a reward may be difficult. For example in the implementation the reward was calculated using a heuristic function which could also be made by the learning agent. On the other hand, in most smart environment applications the ultimate goal is to increase human comfort or efficiency and thus the rewards should be deduced by observing human behaviour.

The described test was able to demonstrate the learning capabilities of the decision maker. However, there were some state–action pairs whose expected rewards were not fully logical compared to other slightly different pairs. More test runs could have revealed if the situation remains or if it is because of random occurrences. For example, there were some random rewards of 0 for some pairs. These decrease the expected reward for the pair by ten per cent, which is the minimum value of the alpha variable.

The exploration strategy of the decision maker could probably also be improved in this case. Because the rate of exploration decreased so slowly, some clearly inferior actions were chosen relatively many times later in the experiment. It would be worth testing with a strategy with a bigger decay parameter or a strategy that decreases first slowly and after a few hundred experiments decreases more rapidly. However, one characteristic of smart environments is dynamics. New devices may occur and disappear and therefore the learning agent should be able to learn and constantly recalculate its model of the environment.

6.3. Summary of Results and Comparison to Other Work

The most important result of this thesis is the model for using machine learning in a smart environment to achieve adaptive control which was presented in Chapter 4. Although the model was only validated using the Smart-M3 IOP, the model itself is more generic and it would probably also work equally well with other kind of interoperability solutions.

There has been other work done regarding the use of machine learning or a subconcept of it in smart environments. This section compares the handling of the subject in them to the results of this thesis. The findings are also compared to the results in the area of adaptive systems.

In the MavHome environment machine learning techniques have been in use for quite a long time. Already in 2002, Das et al. [68] described how prediction algorithms were used in MavHome. Their work concentrates on creating a well-working smart home architecture and therefore they only describe the things that are of interest in the MavHome domain. In 2005, Das and Cook [69] also described the need for reinforcement learning in decision making in smart homes. However, in this thesis other approaches chosen in other projects were also considered and the use of machine learning is presented in a more general form.

The work of Fernandez-Montes et al. [70] has many similar characteristics to this thesis. They also propose a loop structure for smart environments. Their loop consists of three phases: *perception*, *reasoning* and *acting*. These main tasks are further divided into different sub-tasks. In their architecture, all the learning capabilities are included in the reasoning phase in which many different learning tasks are contained in a learning agent. There is no validation or justification presented for the learning task in the paper.

A loop structure similar to the one visible in Figure 7 is also common in other kind of adaptive systems. For example the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop known from the area of autonomic computing [71] is an example of this. This loop consists of monitoring, analysis, planning and execution phases [71, 72]. An autonomic element with a MAPE-K loop is depicted in Figure 22.

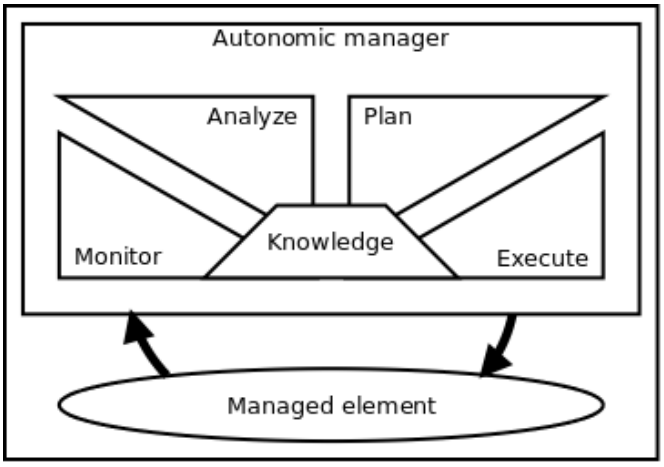


Figure 22. The structure of an autonomic element with a MAPE-K control loop.

The *managed element* can be a hardware or software resource. It has some kind of an internal state that can be observed (through some sensors) and altered (through effectors or actuators). The *autonomic manager* does the required adapting of the element according to some predefined goals. The manager contains the aforementioned MAPE-K loop and uses it to help identify adaptation needs. [71]

The autonomic manager monitors the managed element and its environment constantly and analyses the gathered information. Based on the analysis, it plans the needed actions and executes them. The knowledge can be, for example, an architectural model of the managed element or system, a set of ECA (event-condition-action) rules or utility information and this helps in predicting and understanding the changes in the state of the system. Knowledge is also necessary in the planning phase to determine the best possible actions. [71, 72]

Salehia and Tahvildari call the loop the 'adaptation loop' and name the phases as monitoring, detecting, deciding and acting processes [6]. Their work concerns *self-adaptive software* which is a sub-concept of autonomic computing [6]. There is a rough correspondence between these loops and the machine learning loop in smart environments presented in Figure 7. Monitoring means simply gathering the information, as depicted by the arrow coming from the smart environment. Analysis (or detecting) is done by the pre-processing and prediction elements. The optimisation element does

the adaptation planning (or deciding) and device controllers execute the adaptations (acting process).

All in all, the results of this thesis indicate that machine learning can have an essential role in improving user experience in smart environments. It is also possible to use machine learning in environments using Smart-M3 IOP. However, one problem is the control and cooperation of devices. According to the DIEM white paper [5]: 'The Smart-M3 is meant for opening the information. It does not guarantee the performance and it is not meant for sending commands between devices (or KPs). That kind of interoperability should be implemented using service level capabilities.' The idea behind this statement is that the KPs cannot require different KPs to react in any way to the information they share. However, it is the opinion of the author of this thesis that inter-operability *requires* that other agents react in an understandable way to actions. Additionally, the experiments made for this thesis show that in a controlled environment it is possible to deliver commands using Smart-M3 IOP. However, the ontologies require very careful design to make them suitable for passing commands. Also, the responsibilities about inserting and removing different RDF statements should be defined carefully.

6.4. Future Work

There is still much work to be done in both the areas of machine learning and smart environments. Smart environments can benefit from advances in machine learning and they offer a good application domain in which to develop new learning techniques.

The machine learning model described in Chapter 4 needs more validation. The demonstrations in this thesis validated only two parts of the framework in simplified problem areas. In the future, a real smart environment control loop with different learning solutions should be created in order to validate the whole model. However, this kind of evaluation needs much more infrastructure than was available for this thesis: notably different sensors and actuators with which the state of the environment can be modelled accurately and changed would be needed.

The immaturity of currently available components implementing Smart-M3 IOP caused some unnecessary problems in implementing the demonstrations. However, the basic idea behind the inter-operability solution seems to be working. There has been much valuable work done in making the adoption of Smart-M3 IOP easier, yet it is also important to work on the reliability, stability and scalability of the core components of Smart-M3 IOP. In addition, the comments about Smart-M3 IOP and SIB implementations stated earlier in this chapter should be taken into account while improving the platform or creating new implementations.

7. Conclusions

This thesis described the general uses of machine learning techniques in existing smart environment projects. There were four different categories of machine learning uses found:

1. Detection, in which the goal is to find some characteristics of objects which allow them to be grouped into different categories without explicit labelling.

2. Recognition, in which the goal is to classify an object or an event to a predefined category.

3. Prediction, in which the goal is to find a model of temporal relations between certain measurements or events.

4. Optimisation, in which the goal is to maximise the long-term rewards by making suitable decisions in different situations.

These uses were based on an investigation of published works. Various machine learning techniques were also presented to help in understanding the characteristics of techniques used for different problems.

A loop structure for autonomically managing devices in the smart environment using machine learning was proposed. The loop consists of the aforementioned categories with the optimisation category divided into decision making and control problems.

Two demonstrations were implemented to validate the resulting model. In the first one, a latency predictor, it attempted to generate a model of the effect of the number of joined KPs, the number of triples in the database and the number of subscriptions to the time it takes to make a query to the SIB. The latency predictor used a neural network with a backpropagation-based learning algorithm for prediction. It was able to predict the latencies better than simpler reference predictors in cases where the change in the amounts of subscriptions and the succeeding change in the latency were rapid. However, it also made false deductions about the relationships between inputs and outputs in some situations.

The second demonstration was a decision maker whose task was to choose a filter device for a video-streaming application. It used an implementation of the Q learning algorithm with a look-up table to estimate the rewards for different choices. The rewards were calculated using the delay of completing the task and the mean and standard deviation of the processor utilisations of the devices in the smart environment as parameters. The learning agent did not know this reward function. It was able to fill the table with reasonable values although some values seemed somewhat illogical compared to other values. It is probable that by using the look-up table created by the learning agent the decision maker could make optimal decisions for these kinds of tasks.

Machine learning techniques are now, and in the future, essential to enhance the behaviour of smart environments. The most important work is to develop machine learning solutions to the problems described in this thesis that are adaptive and easy to use in different domains.

References

[1] Cook, D.J. & Das, S.K. (2007) How smart are our environments? An updated look at the state of the art. Pervasive and Mobile Computing 3, pp. 53–73. Design and Use of Smart Environments.

[2] Mozer, M.C. (1998) The Neural Network House: An Environment that Adapts to its Inhabitants. In: Proc. AAAI Spring Symposium on Intelligent Environments.

[3] Cook, D.J., Youngblood, M. & Das, S.K. (2006) A Multi-agent Approach to Controlling a Smart Environment, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 4008. pp. 165–182.

[4] Holmes, A., Duman, H. & Pounds-Cornish, A. (2002) The iDorm: Gateway to Heterogeneous Networking Environments. In: International ITEA Workshop on Virtual Home Environments, pp. 30–37.

[5] Soininen, J.P., Liuha, P., Lappeteläinen, A., Honkola, J., Främling, K. & Raisamo, R. (2010), Tivit/DIEM project, White paper. URL: `http://www.tivit.fi/fi/dokumentit/64/DIEM%20whitepaper.pdf`.

[6] Salehie, M. & Tahvildari, L. (2009) Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems 4, pp. 14:1–14:42.

[7] Mitchell, T.M. (1997) Machine learning. McGraw-Hill, New York.

[8] Cook, D.J. & Das, S.K. (2005) Smart Environments: Technologies, Protocols, and Applications. John Wiley, Hoboken, NJ.

[9] Hermann, F., Blach, R., Janssen, D., Klein, T., Schuller, A. & Spath, D. (2009) Challenges for User Centered Smart Environments, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 5612. pp. 407–415.

[10] Mozer, M.C. (2005) Lessons from an Adaptive Home. In: D.J. Cook & S.K. Das (eds.) Smart Environments: Technologies, Protocols, and Applications, John Wiley, pp. 273–294.

[11] Youngblood, G.M., Cook, D.J. & Holder, L.B. (2005) Managing Adaptive Versatile environments. Pervasive and Mobile Computing 1, pp. 373–403. Special issue on PerCom 2005.

[12] Cook, D.J. (2008) Artificial Intelligence on the Body, in the Home, and Beyond. In: BodyNets '08: Proceedings of the ICST 3rd international conference on Body area networks, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, Belgium, pp. 1–6.

[13] Jakkula, V.R., Crandall, A.S. & Cook, D.J. (2009) Enhancing Anomaly Detection Using Temporal Pattern Discovery. In: Advanced Intelligent Environments, Springer US, pp. 175–194.

[14] Remagnino, P., Hagras, H., Monekosso, N. & Velastin, S. (2005) Ambient Intelligence: A Gentle Introduction. In: Ambient Intelligence, Springer New York, pp. 1–14.

[15] Hagras, H., Callaghan, V., Colley, M., Clarke, G., Pounds-Cornish, A. & Duman, H. (2004) Creating an Ambient-Intelligence Environment Using Embedded Agents. IEEE Intelligent Systems 19, pp. 12–20.

[16] Hagras, H., Doctor, F., Callaghan, V. & Lopez, A. (2007) An Incremental Adaptive Life Long Learning Approach for Type-2 Fuzzy Embedded Agents in Ambient Intelligent Environments. IEEE Transactions on Fuzzy Systems 15, pp. 41–55.

[17] Duman, H., Hagras, H. & Callaghan, V. (2010) A Multi-Society-Based Intelligent Association Discovery and Selection for Ambient Intelligence Environments. ACM Transactions on Autonomous and Adaptive Systems 5, pp. 1–34.

[18] Tawil, E. & Hagras, H. (2009) An Adaptive Genetic-Based Incremental Architecture for the On-Line Coordination of Embedded Agents. Cognitive Computation 1, pp. 300–326.

[19] Hagras, H., Callaghan, V., Colley, M. & Clarke, G. (2003) A hierarchical fuzzy-genetic multi-agent architecture for intelligent buildings online learning, adaptation and control. Information Sciences 150, pp. 33–57.

[20] Reinisch, C., Kofler, M.J. & Kastner, W. (2010) ThinkHome: A Smart Home as Digital Ecosystem. In: Proceedings of 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST 2010), pp. 256–261.

[21] MIT Project Oxygen (accessed 24.5.2010). URL: `http://oxygen.csail.mit.edu/`.

[22] DreamSpace (accessed 24.5.2010). URL: `http://www.research.ibm.com/natural/dreamspace/`.

[23] de Ruyter, B., van Loenen, E. & Teeven, V. (2007) User Centered Research in ExperienceLab, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 4794. pp. 305–313.

[24] Brumitt, B., Meyers, B., Krumm, J., Kern, A. & Shafer, S. (2000) EasyLiving: Technologies for Intelligent Environments, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 1927. pp. 97–119.

[25] Intille, S.S., Larson, K., Beaudin, J.S., Nawyn, J., Tapia, E.M. & Kaushik, P. (2005) A Living Laboratory for the Design and Evaluation of Ubiquitous Computing Technologies. In: CHI '05 extended abstracts on Human factors in computing systems, ACM, New York, NY, USA, pp. 1941–1944.

[26] Intille, S.S., Larson, K., Tapia, E.M., Beaudin, J.S., Kaushik, P., Nawyn, J. & Rockinson, R. (2006) Using a Live-In Laboratory for Ubiquitous Computing Research, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 3968. pp. 349–365.

[27] Logan, B., Healey, J., Philipose, M., Tapia, E.M. & Intille, S. (2007) A Long-Term Evaluation of Sensing Modalities for Activity Recognition, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 4717. pp. 483–500.

[28] Duda, R.O., Hart, P.E. & Stork, D.G. (2001) Pattern Classification. John Wiley & Sons, New York, second ed.

[29] Michalski, R.S., Carbonell, J.G. & Mitchell, T.M. (eds.) (1983) Machine Learning: An Artificial Intelligence Approach. Morgan Kaufmann, Los Altos, Calif.

[30] van Someren, M. & Urbančič, T. (2006) Applications of Machine Learning: Matching Problems to Tasks and Methods. Knowledge Engineering Review 20, pp. 363–402.

[31] Quinlan, J. (1993) C4.5: Programs for Machine Learning. Morgan Kaufmann.

[32] Breiman, L. (2001) Random Forests. Machine learning 45, pp. 5–32.

[33] Russell, S. & Norvig, P. (2003) Artificial Intelligence: A Modern Approach. Prentice-Hall, Upper Saddle River, New Jersey, second ed.

[34] Haykin, S. (1999) Neural Networks: A Comprehensive Foundation. Prentice-Hall, Upper Saddle River, New Jersey, second ed.

[35] Fahlman, S.E. & Lebiere, C. (1991) The Cascade-Correlation Learning Architecture. Tech. Rep. CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[36] Rabiner, L.R. & Juang, B.H. (1986) An Introduction to Hidden Markov Models. IEEE ASSP Magazine 3, pp. 4–16.

[37] Kotsiantis, S.B. (2007) Supervised Machine Learning: A Review of Classification Techniques. Informatica 31, pp. 249–268.

[38] Kotsiantis, S.B., Zaharakis, I.D. & Pintelas, P.E. (2006) Machine learning: a review of classification and combining techniques. Artificial Intelligence Review 6, pp. 159–190.

[39] Watkins, C. (1989) Learning from Delayed Rewards. Ph.D. thesis, King's College, Cambridge, England.

[40] Dietterich, T.G. (2003) Machine Learning (accessed 17.5.2010). In: Nature Encyclopedia of Cognitive Science, MacMillan, London. URL: `http://www.cs.utsa.edu/~bylander/cs6243/nature-ecs-machine-learning.pdf`.

[41] Könönen, V. (2004) Multiagent Reinforcement Learning in Markov Games: Asymmetric and Symmetric Approaches. Ph.D. thesis, Helsinki University of Technology, Deparment of Computer Science and Engineering, Espoo, Finland.

[42] Sumathi, S. & Sivanandam, S. (2006) Introduction to Data Mining and its Applications. Studies in Computational Intelligence, Springer Berlin / Heidelberg.

[43] Berkhin, P. (2006) A Survey of Clustering Data Mining Techniques. In: Grouping Multidimensional Data, Springer Berlin Heidelberg, pp. 25–71.

[44] Patcha, A. & Park, J.M. (2007) An overview of anomaly detection techniques: Existing solutions and latest technological trends. Computer Networks 51, pp. 3448–3470.

[45] Vilalta, R. & Ma, S. (2002) Predicting Rare events In Temporal Domains. In: Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM 2002., pp. 474–481.

[46] Weiss, G.M. & Hirsh, H. (1998) Learning to Predict Rare Events in Event Sequences. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, AAAI Press, pp. 359–363.

[47] Laxman, S., Tankasali, V. & White, R.W. (2008) Stream Prediction Using A Generative Model Based On Frequent Episodes In Event Sequences. In: KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, pp. 453–461.

[48] Ipek, E., de Supinski, B.R., Schulz, M. & McKee, S.A. (2005) An Approach to Performance Prediction for Parallel Applications, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 3648. pp. 196–205.

[49] Han, J., Cheng, H., Xin, D. & Yan, X. (2007) Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery 15, pp. 55–86.

[50] Van Kasteren, T., Noulas, A., Englebienne, G. & Kröse, B. (2008) Accurate Activity Recognition in a Home Setting. In: UbiComp '08: Proceedings of the 10th international conference on Ubiquitous computing, ACM, New York, NY, USA, pp. 1–9.

[51] Mühlenbrock, M., Brdiczka, O., Snowdon, D. & Meunier, J.L. (2004) Learning to Detect User Activity and Availability from a Variety of Sensor Data. In: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004., pp. 13–22.

[52] Hafner, R. & Riedmiller, M. (2007) Neural Reinforcement Learning Controllers for a Real Robot Application. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 07), pp. 2098–2103.

[53] Prothmann, H., Branke, J., Schmeck, H., Tomforde, S., Rochner, F., Hähner, J. & Müller-Schloer, C. (2009) Organic traffic light control for urban road networks. International Journal of Autonomous and Adaptive Communications Systems 2, pp. 203–225.

[54] Lappeteläinen, A., Tuupola, J.M., Palin, A. & Eriksson, T. (2008) Networked systems, services and information: The ultimate digital convergence. In: 1st International Network on Terminal Architecture Conference (NoTA2008).

[55] Evesti, A., Eteläperä, M., Kiljander, J., Kuusijärvi, J., Purhonen, A. & Stenudd, S. (2009) Semantic Information Interoperability in Smart Spaces. In: Proceedings of the 8th International Conference on Mobile and Ubiquitous Multimedia, ACM International Conference Proceedings Series, pp. 158–159.

[56] Gruber, T.R. (1995) Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Journal of Human Computer Studies 43, pp. 907–928.

[57] Motik, B., Parsia, B. & Patel-Schneider, P.F. (2009) OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C Recommendation, W3C. URL: http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/.

[58] Carroll, J.J. & Klyne, G. (2004) Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, W3C. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[59] Guha, R.V. & Brickley, D. (2004) RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, W3C. URL: http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[60] Smart-M3 at SourceForge.net (accessed 24.5.2010). URL: http://sourceforge.net/projects/smart-m3/.

[61] Lassila, O. (2007) Programming Semantic Web Applications: A Synthesis of Knowledge Representation and Semi-Structured Data. Ph.D. thesis, Helsinki University of Technology, Deparment of Computer Science and Engineering, Espoo, Finland.

[62] Kiljander, J. (2010) Reference Implementation of Interoperable Entity for Smart Environments. Master's thesis, University of Oulu, Department of Electrical and Information Engineering, Finland.

[63] Igel, C., Glasmachers, T. & Heidrich-Meisner, V. (2008) Shark. Journal of Machine Learning Research 9, pp. 993–996.

[64] Igel, C. & Hüsken, M. (2003) Empirical evaluation of the improved Rprop learning algorithms. Neurocomputing 50, pp. 105–124.

[65] Taymans, W., Baker, S., Wingo, A., Bultje, R.S. & Kost, S., GStreamer Application Development Manual (0.10.29) (accessed 12.5.2010). URL: http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf.

[66] van Dorst, W., BogoMips mini-Howto (accessed 12.5.2010). URL: http://www.clifton.nl/index.html?bogomips.html.

[67] Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T. & Schmidhuber, J. (2010) PyBrain. Journal of Machine Learning Research .

[68] Das, S.K., Cook, D.J., Battacharya, A., Heierman, III, E.O. & Lin, T.Y. (2002) The Role of Prediction Algorithms in the MavHome Smart Home Architecture. Wireless Communications, IEEE 9, pp. 77–84.

[69] Das, S.K. & Cook, D.J. (2005) Designing Smart Environments: A Paradigm Based on Learning and Prediction, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol. 3776. pp. 80–90.

[70] Fernandez-Montes, A., Ortega, J.A., Alvarez, J.A. & Gonzalez-Abril, L. (2009) Smart Environment Software Reference Architecture. In: Fifth International Joint Conference on INC, IMS and IDC, 2009. NCM '09., pp. 397–403.

[71] Kephart, J.O. & Chess, D.M. (2003) The Vision of Autonomic Computing. IEEE Computer 36, pp. 41–50.

[72] Huebscher, M.C. & McCann, J.A. (2008) A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv. 40, pp. 1–28.

Series title, number and report code of publication

VTT Publications 751
VTT-PUBS-751

Author(s)
Sakari Stenudd

Title

Using machine learning in the adaptive control of a smart environment

Abstract

The purpose of this thesis is to study the possibilities and need for utilising machine learning in a smart environment. The most important goal of smart environments is to improve the experience of their inhabitants. This requires adaptation to the behaviour of the users and the other changing conditions in the environment. Hence, the achievement of functional adaptation requires finding a way to change the behaviour of the environment according to the changed user behaviour and other conditions. Machine learning is a research area that studies the techniques which make it possible for software agents to improve their operation over time.

The research method chosen in this thesis was to review existing smart environment projects and to analyse the usages of machine learning within them. Based upon these uses, a model for using machine learning in a smart environment was created. As a result, four different categories of machine learning in smart environments were identified: *prediction*, *recognition*, *detection* and *optimisation*. When deployed to the environment, these categories form a clear loop structure in which the outputs of previous learning agents serve as inputs for the next agents, which ultimately enables the making of changes to the environment according to its current state. This kind of loop is called a *control loop* in adaptive systems.

To evaluate the suitability of the model for using machine learning in a smart environment, two demonstrations were carried out in an environment using a Smart-M3 inter-operability platform, both utilising machine learning in one of the above-discussed categories. In the first experiment neural networks were used to predict query latencies in different situations in the environment. The predictions of the network were compared to the outputs of two simpler models. The results showed that the neural network approach was capable of adapting to rapid changes more quickly. However, it also made more false assumptions about the impact of the different parameters.

The second experiment belongs to the optimisation category. In this experiment a decision maker was implemented for a resource allocation problem in a distributed multi-media streaming application. It used reinforcement learning with a look-up table and an implementation of the Q-learning algorithm. After the learning period the agent was capable of making optimal decisions.

The experiments confirm that it is suitable to use the model described in this thesis in smart environments. The model includes the most important uses of machine learning and it is consistent with other results in the areas of smart environments and self-adaptive software.

ISBN
978-951-38-7420-9 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN	Project number
VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	

Date	Language	Pages
December 2010	English, finnish abstr.	75 p.

Name of project	Commissioned by

Keywords	Publisher
Smart space, inter-operability, control loop, adaptive systems, self-adaptive software, reinforcement learning, Smart-M3 IOP	VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 751
VTT-PUBS-751

Tekijä(t)
Sakari Stenudd

Nimeke

Koneoppimisen käyttö äly-ympäristön mukautuvassa ohjauksessa

Tiivistelmä

Opinnäytetyöni tarkoitus on tutustua koneoppimisen käyttömahdollisuuksiin ja -tarpeisiin älykkäässä ympäristössä. Älykkäiden ympäristöjen tärkein päämäärä on niiden käyttäjien käyttökokemuksen parantaminen. Tämä vaatii mukautumista käyttäjien käytökseen sekä muihin muuttuviin tilanteisiin ympäristössä. Mukautumisen saavuttamiseksi tarvitaan tapa muuttaa ympäristön toimintaa tapahtuvien muutosten mukaan. Koneoppiminen on tutkimusalue, joka käsittelee sellaisia tekniikoita, joita käyttäen ohjelmistoagentit voivat parantaa toimintaansa ajan kuluessa.

Opinnäytetyön alussa tutustutaan olemassa oleviin äly-ympäristöprojekteihin ja tarkastellaan niissä käytettyjä koneoppimismenetelmiä. Käytettyihin menetelmiin perustuen esitetään malli, joka kuvaa, miten koneoppimismenetelmiä voidaan käyttää älykkäissä ympäristöissä. Malli sisältää neljä eri koneoppimistyyppiä: *havainnointi*, *tunnistaminen*, *ennustaminen* ja *optimointi*. Kun näitä tyyppejä käytetään äly-ympäristössä, ne muodostavat selkeän silmukkarakenteen, jossa seuraavat oppivat agentit voivat käyttää edellisten tuloksia. Tämä mahdollistaa lopulta sen, että ympäristöön voidaan tehdä muutoksia sen nykyisen tilan perusteella. Tällaista rakennetta kutsutaan mukautuvien järjestelmien alueella nimellä *ohjaussilmukka*.

Jotta voitaisiin arvioida luodun mallin soveltuvuutta, luotiin kaksi mallin osa-aluetta käyttävää demonstraatiota käyttäen Smart-M3-yhteentoimivuusalustaa. Ensimmäisessä toteutuksessa käytettiin neuroverkkoja ennustamaan kyselyjen viivettä erilaisissa äly-ympäristön tilanteissa. Neuroverkon ennusteita verrattiin kahden yksinkertaisemman mallin tuloksiin. Testit osoittivat, että neuroverkkomenetelmä pystyi mukautumaan nopeisiin muutoksiin aiemmin, mutta se teki myös joitakin vääriä olettamuksia eri parametrien vaikutuksesta tulokseen.

Toinen koe kuuluu optimointiluokkaan. Siinä toteutettiin päätöksentekijäohjelma, jonka tuli ratkaista resurssien kohdentamisongelma hajautetussa multimedian suoratoisto-ohjelmassa. Päätöksentekijässä sovellettiin vahvistusoppimistekniikkaa käyttäen hakutaulukkoa ja Q-oppimisen toteutusta. Oppimisjakson jälkeen agentti pystyi tekemään optimaalisia päätöksiä suurimman osan ajasta.

Tehdyt kokeet osoittivat, että työssä kuvattu malli sopii käytettäväksi älykkäissä ympäristöissä. Malli kattaa tärkeimmät koneoppimisen käyttökohteet ja on yhtäpitävä muiden tulosten kanssa, jotka on saatu äly-ympäristöjen ja mukautuvien ohjelmistojen alueella.

ISBN
978-951-38-7420-9 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN	Projektinumero
VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	

Julkaisuaika	Kieli	Sivuja
Joulukuu 2010	Englanti, suom. tiiv.	75 s.

Projektin nimi	Toimeksiantaja(t)

Avainsanat	Julkaisija
Smart space, inter-operability, control loop, adaptive systems, self-adaptive software, reinforcement learning, Smart-M3 IOP	VTT PL 1000, 02044 VTT Puh. 020 722 4520 Faksi 020 722 4374

VTT PUBLICATIONS

ISBN 978-951-38-7420-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)